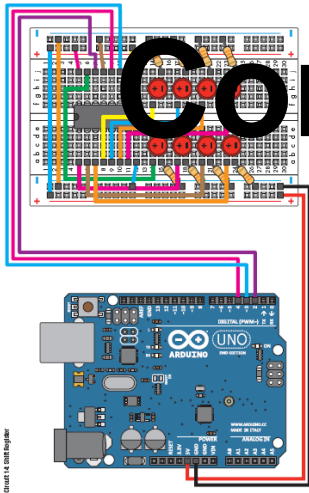
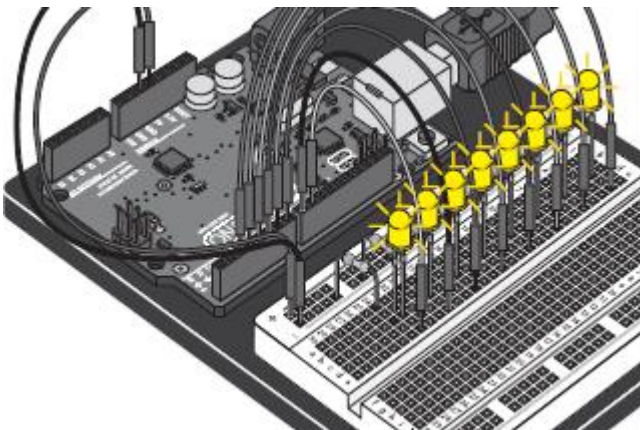
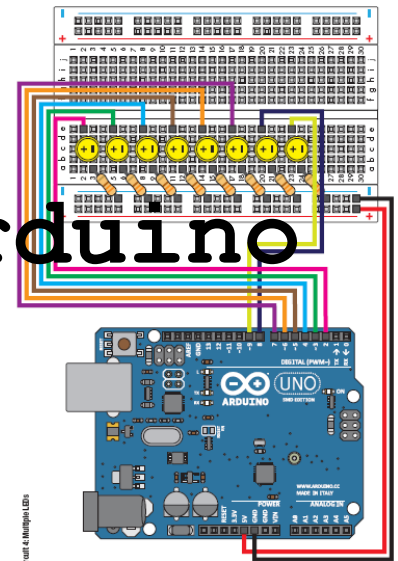


# Foundations of Computer Science Workbook



## Exploring the Arduino





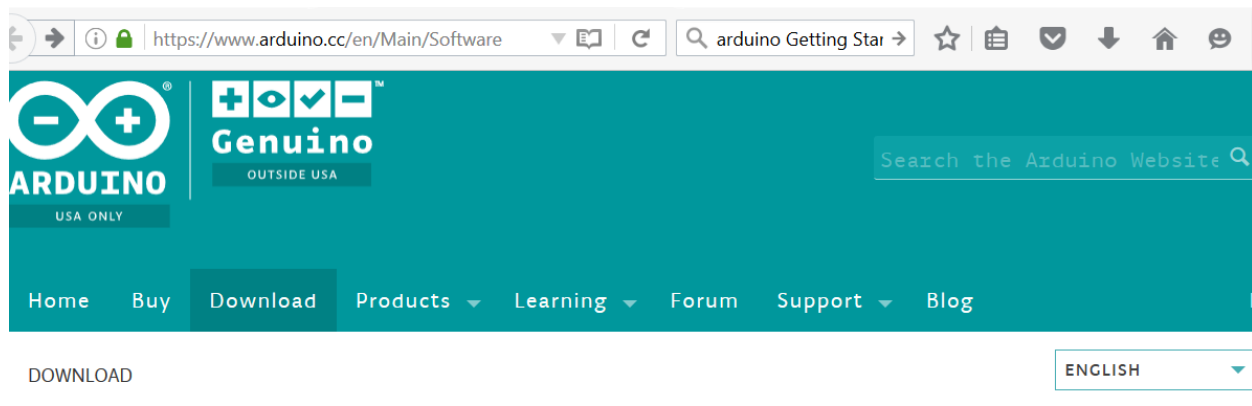
# Preface

# General Information


## Getting Started with Arduino Software

Go to the web site ==>>>

<https://www.arduino.cc/en/Main/Software>



## Download the Arduino Software



**ARDUINO 1.6.11**  
The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. It runs on Windows, Mac OS X, and Linux. The environment is written in Java and based on Processing and other open-source software. This software can be used with any Arduino board.  
Refer to the [Getting Started](#) page for Installation instructions.

**Windows** Installer  
**Windows** ZIP file for non admin install

**Mac OS X** 10.7 Lion or newer

**Linux** 32 bits  
**Linux** 64 bits  
**Linux** ARM (experimental)

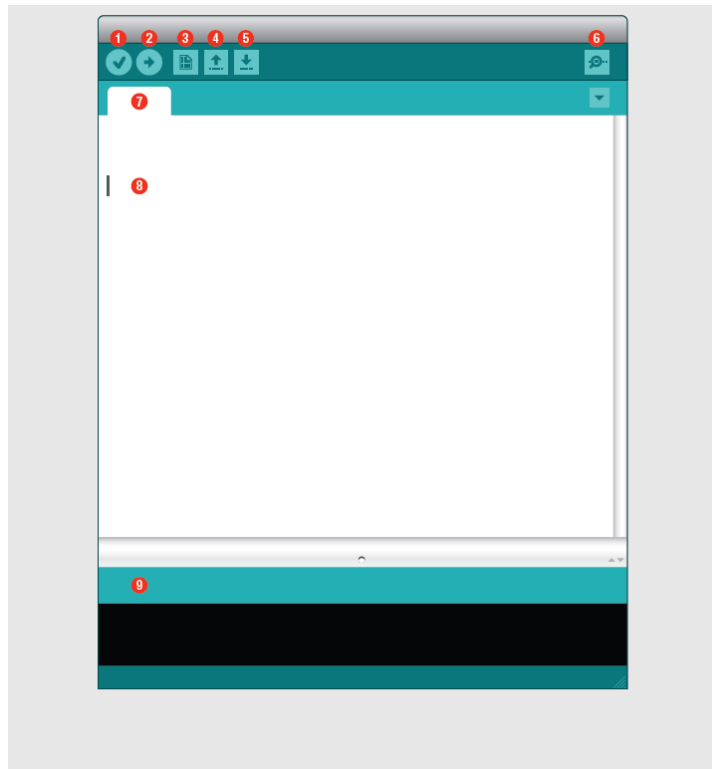
[Release Notes](#)  
[Source Code](#)



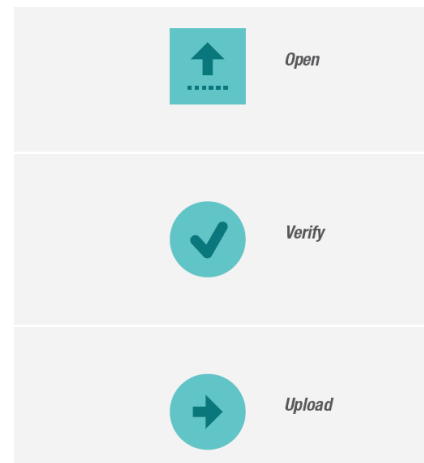
Click on “Getting Started” on this web page. Then follow the Installation instructions for your particular system type.

# The Arduino IDE

## “Interactive Development Environment”



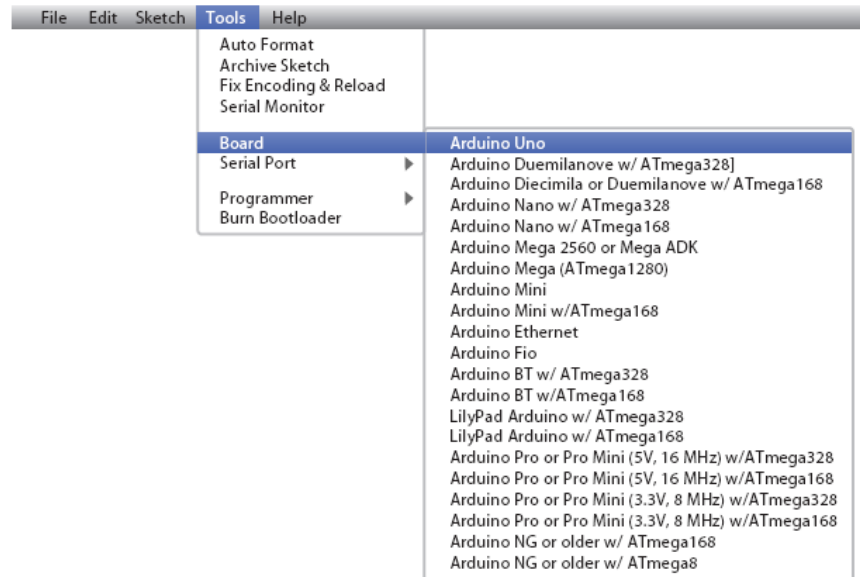
### Three Most Important Commands



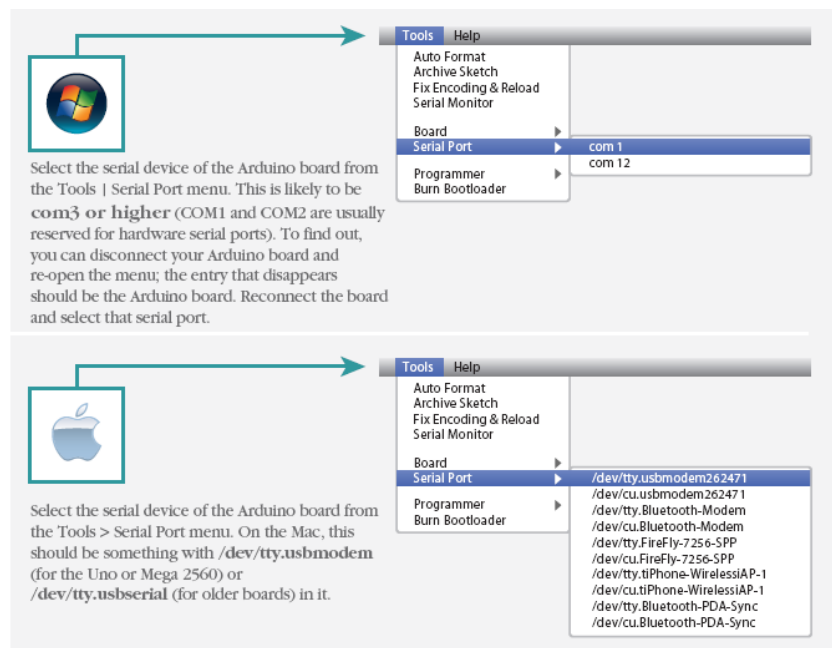
#### GUI (Graphical User Interface)

- 1 **Verify:** Compiles and approves your code. It will catch errors in syntax (like missing semi-colons or parentheses). // See Diagram Below
- 2 **Upload:** Sends your code to the Arduino board. When you click it, you should see the lights on your board blink rapidly. // See Diagram Below
- 3 **New:** This button opens up a new code window tab.
- 4 **Open:** This button will let you open up an existing sketch. // See Diagram Below
- 5 **Save:** This saves the currently active sketch.
- 6 **Serial Monitor:** This will open a window that displays any serial information your Arduino is transmitting. It is very useful for debugging.
- 7 **Sketch Name:** This shows the name of the sketch you are currently working on.
- 8 **Code Area:** This is the area where you compose the code for your sketch.
- 9 **Message Area:** This is where the IDE tells you if there were any errors in your code.

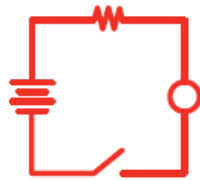
## Selecting Your Board



## Selecting your Serial Device “com port”



# Circuits



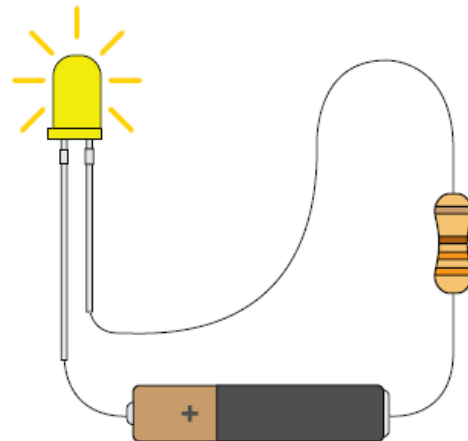
## *What is an Electrical Circuit?*

A circuit is basically an electronics loop with a starting point and an ending point - with any number of components in between. Circuits can include resistors, diodes, inductors, sensors of all sizes and shapes, motors, and any other handful of hundreds of thousands of components.

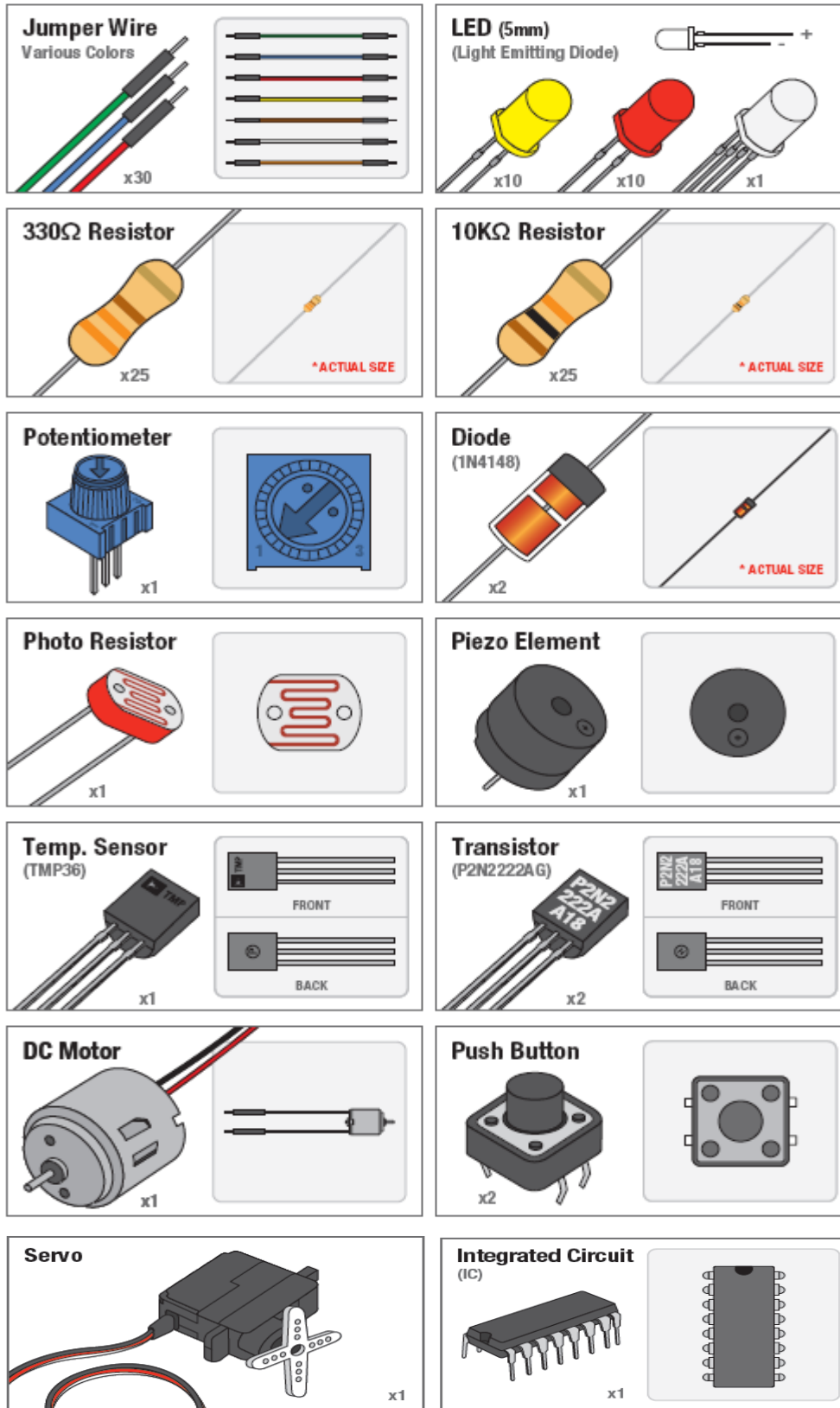
Circuits are usually divided into three categories - analog circuits, digital circuits, or mixed-signal circuits. In this guide, you will explore all three sets of circuits.

## *The World Runs on Circuits:*

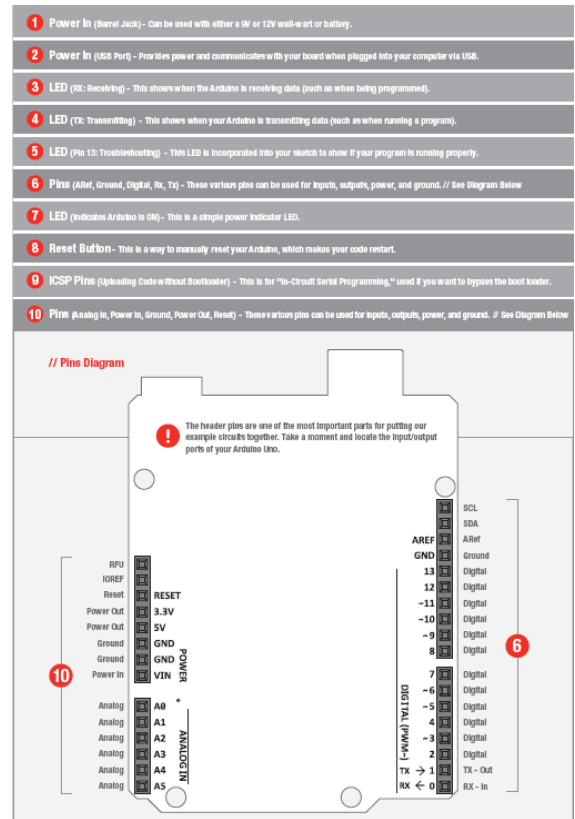
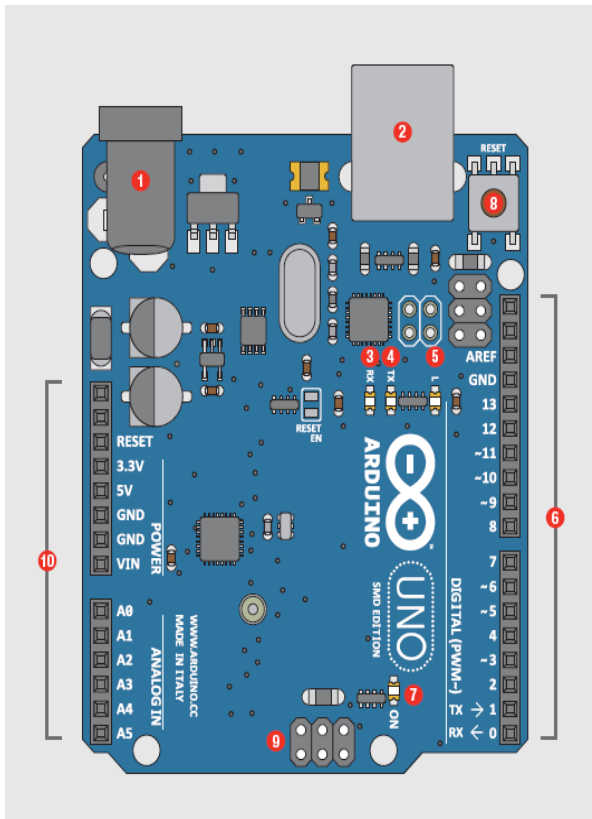
Everywhere you look, you'll find circuits. The cell phone in your pocket, the computer that controls your car's emissions system, your video game console - all these things are chock full of circuits. In this guide, you'll experiment with some simple circuits and learn the gist of the world of embedded electronics.



## Parts

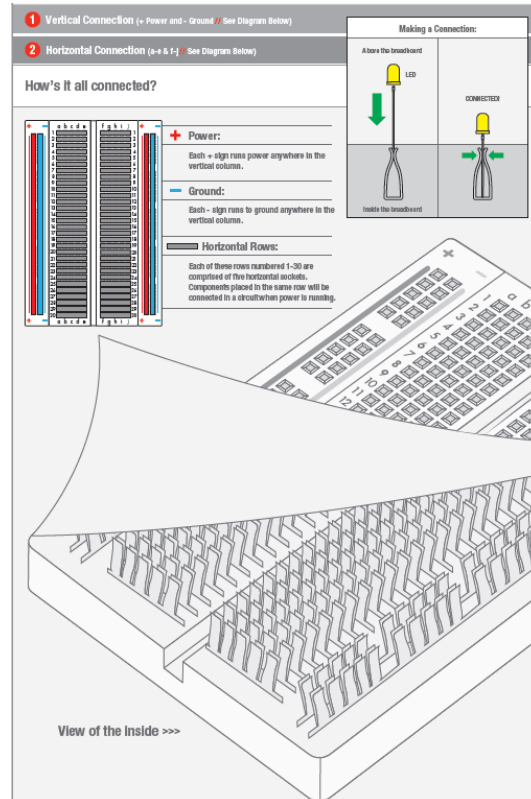
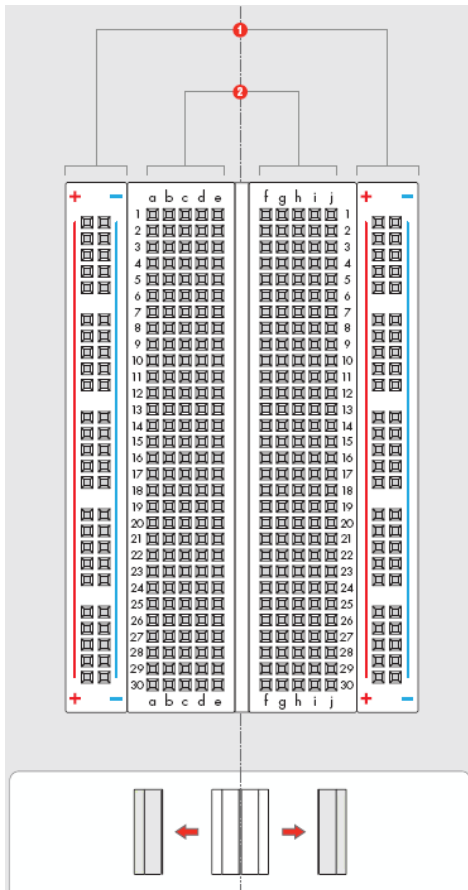


# The Arduino Uno





# The Bread Board



## TABLE OF CONTENTS

### UNIT 1 THE BASICS

---

THE BASICS	9
HOMEWORK	15
ADD-ONS & PROJECTS	17

### UNIT 2 VARIABLES

---

VARIABLES	18
HOMEWORK	21
PROJECTS	22

### UNIT 3 CODING STYLE

---

CODING STYLE	23
HOMEWORK	27

### UNIT 4 ARITHMETIC & SERIAL MONITOR

---

ARITHMETIC & SERIAL MONITOR	29
HOMEWORK	37
PROJECTS	43

### UNIT 5 CONTROL STRUCTURES

---

CONTROL STRUCTURES	44
HOMEWORK	52
PROJECTS	56

### UNIT 6 YOUR OWN FUNCTIONS & SCOPE

---

YOUR OWN FUNCTIONS & SCOPE	57
HOMEWORK	68
PROJECTS	73

### UNIT 7 WORKING WITH DATA STRUCTURES - ARRAYS

---

WORKING WITH DATA STRUCTURES - ARRAYS	78
HOMEWORK	79
PROJECTS	82

## **UNIT 8    STRINGS AND PROGRAMMING BY INTENTION**

---

<b>STRINGS AND PROGRAMMING BY INTENTION</b>	<b>83</b>
HOMEWORK	98
PROJECTS	100

## **UNIT 9    OBJECT ORIENTED C++**

---

<b>OBJECT ORIENTED C++</b>	<b>78</b>
HOMEWORK	79
PROJECTS	82

## **UNIT 10    POINTERS IN C++**

---

<b>POINTERS IN C++</b>	<b>78</b>
HOMEWORK	79
PROJECTS	82

# Unit 1     The Basics

It is best to think of a program or sketch, as programs are called in Arduino, is as *a list of instructions to be carried out in the order that they are written down*. For example, suppose you were to write the following:

```
digitalWrite(13, HIGH);  
delay(500);  
digitalWrite(13, LOW);
```

These three lines each do something. The first line sets the output of pin 13 to HIGH (5 volts). This is the pin with an LED built in to the Arduino board, so at this point the LED would light. The second line would simply wait for 500 milliseconds (half a second) and then the third line would turn the LED back off again. So these three lines would achieve the goal of making the built in LED blink once. Please note that computers are very picky about how we communicate with them. *Punctuation and word choice are critical*.



Let's start by dealing with the *punctuation* and the way words are formed. These are both part of what is termed the **syntax** of the computer language. **Computer languages require you to be extremely precise about syntax**. One of the main rules in **C** is that names for things have to be a single word. That is, they cannot include spaces. `digitalWrite` is the name for a built-in function that comes with the Arduino Interactive Development Environment (IDE). That function will do the job of setting an output pin on the Arduino board to HIGH or LOW. Now, not only do you have to avoid spaces in names, but names are also *case sensitive*. You **must** write `digitalWrite`, not `DigitalWrite` or `Digitalwrite`.

In addition the function `digitalWrite` needs to know which pin to set and whether to set that pin value HIGH or LOW. These two pieces of information are called *arguments or parameters*. They are said to be passed to a function when it is called. Also notice that the parameters for a function *must be enclosed in parentheses and separated by commas*.

The convention we will always follow in our class is to place the opening parenthesis immediately after the last letter of the function's name and to put a space after the comma before the next parameter. This will make it much easier to read our code.

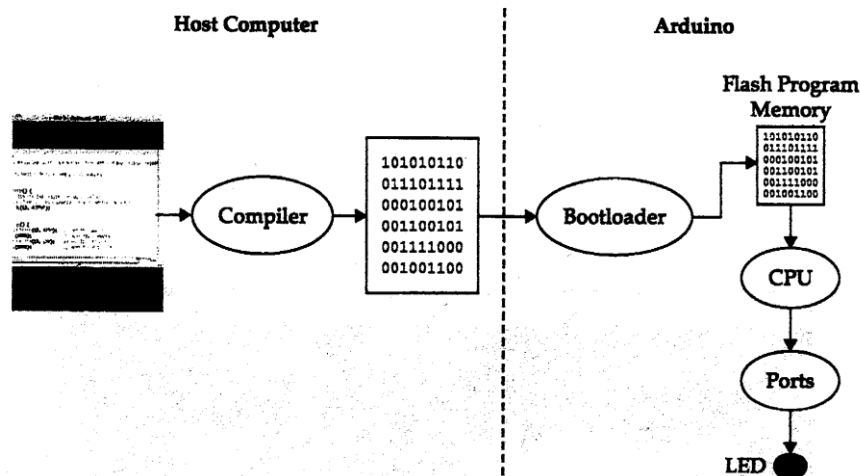
If the function only has one argument, then there is no need for a comma. Notice that each line of code ends with a semicolon. The semicolon marks the end of one command, a bit like the way a period is used at the end of a sentence.


## The Compile and Upload Buttons

Let's find out a bit about what happens when you press the Compile  and then the Upload  button on the Arduino integrated development environment (IDE). These buttons are at the top and on the left side of your programming window.


It is perhaps a little surprising that we can get to this point discussing programming without defining exactly what a programming language is. We can probably recognize an Arduino sketch and have a rough idea of what it is trying to do, but we need to look a bit deeper into how programming language code goes from being words on a page to something that does something real like turning on and off an LED.

Below we summarize the process involved from typing code into the Arduino environment to running the sketch on the Arduino board itself.



The first step is something called compilation. This takes the code you have written and translates it into machine code--the binary language that the Arduino can understand. If you click the Compile/check-verify  button on the Arduino IDE, the IDE will actually attempt to compile the C program

that you have written without trying to send the compiled code to the Arduino. A side-effect of compiling the code is that it is checked to make sure that it conforms to the rules of the C programming language.

When you press the Upload button  on your Arduino IDE, it launches a chain of events that results in your sketch being installed on the Arduino and being running. This is not as straightforward as simply taking the text that you typed into the editor of the IDE and moving it to the Arduino board.

In order for your sketch to compile you must have to have some "boilerplate" code, as it is called; before you can add your own code into a sketch. In Arduino programming the "boilerplate" code takes the form of the "setup" and "loop" functions. They must always be present in your sketch.

You will learn much more about functions later in this workbook, but for now, let's accept that you need this boilerplate code and just adapt your sketch so it will compile.

If the Arduino IDE looks at your coding effort and it has found them to be acceptable it will tell you this by saying "Done Compiling" and reporting the size of the sketch to you have written in bytes. The IDE will also tell you that the maximum size that any sketch can be is 32,256 bytes. Most usually this will be more than enough for any program we will write in this class.

Let's examine this boilerplate code. This code will form the starting point for every sketch that you will ever write. There are some new words here. For example there is the word `void` and some curly braces `{ }`.

The line `void setup()` means that you are defining a function called `setup`. In the Arduino IDE, some functions are already defined for you, such as `digitalWrite` and `delay`, you can also define others for yourself. `setup` and `loop` are two functions that must be in every one of your sketches, whether or not you fill them in or define them for yourself.

The important thing to understand is that here you are not calling `setup` or `loop`. You are actually creating these functions so that the Arduino system itself can call them. This can be a difficult concept to grasp. *Just remember that most functions you will call, but these two functions will be called by the **Arduino IDE itself**.*

Going back to the word `void`. The two functions (`setup` and `loop`) do not return a value as some functions do. If a function does not return a value when called they are said to be void and they must be declared so using the void key word. Imagine that you have a function like `sqrt()` (square root).

When called this function would return a value. For example if you took the `sqrt(4.0)` the number `2.0` would be returned. However `setup()` and `loop()` do not return a value, so they are considered `void`.

After the special keyword `void` comes the name of the function and then parentheses to contain any arguments or input into the function. For example the function `sqrt(4.0)` had the argument `4.0`. In the case of `setup` and `loop` there are no arguments, but we still have to include the parentheses there. There is no semicolon after the closing parentheses with these because we are defining a function and not calling it. Inside the curly braces we will need to type in the code that will execute when the Arduino IDE does call those functions. The curly braces and the code in between then are known as a **block of code**.

The reason that Arduino has the two functions **setup** and **loop** is to separate and place the things that *only need to be done once* in setup, when the Arduino starts running its sketch. And then place the things that have to keep *happening continuously* in the loop.

The function `setup` will just be run just once when the sketch starts. Let's add some code that will blink the LED built onto the board. Add the following lines to your sketch as it appears below and then upload them to your board:

```
void setup ()
{
    pinMode(13, OUTPUT);
    digitalWrite(13, HIGH);
}

void loop()
{}
```

The `setup` function itself calls two built-in functions, `pinMode` and `digitalWrite`. You already know about `digitalWrite`, but `pinMode` is new. The function `pinMode` sets a particular pin on your Arduino board to have a mode that is either an INPUT or an OUTPUT. So, turning the LED on is actually a two-stage process. First, you have to set pin 13 to be an output, and second, you need to set that output to be high (5 Volts).

When you run this sketch, on your board you will see that the LED comes on and stays on. This is not very exciting, so let's at least try to make it

flash by turning it on and off in the loop function rather than in the setup function.

You can leave the `pinMode` call in the `setup` function because you only need to call it once. The sketch would still work if you moved it into the `loop`, but there is no need to and it is a good programming habit to do things, only once if you only need to do them once. So modify your sketch so that it looks like this:

```
void setup ()
{
    pinMode(13, OUTPUT);
}

void loop()
{
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
}
```

Compile and upload this sketch and see what happens. It may not be quite what you were expecting. The LED is basically on all the time. Hmm, why should this be?

Let's try stepping through the sketch a line at a time in your head. This process is often very helpful when things just do not seem to make sense:

1. Run setup and set pin 13 to be an OUTPUT.
2. Run loop and set pin 13 to HIGH (LED on).
3. Delay for half a second (500 milliseconds).
4. Set pin 13 to LOW (LED off).
5. **Run loop again**, going back to step 2, and set pin 13 to, HIGH (LED on).

The problem lies between steps 4 and 5. What is happening is that the LED is being turned off, but the very next thing that happens is that it gets turned on again. This happens so quickly that it appears that the LED is on all the time.

The microcontroller chip on the Arduino can perform *16 million instructions per second*. That's not 16 million C language commands, but it is



still very fast. So, our LED will only be off for a few millionths of a second. To our eye the LED will appear to always be on.

To fix the problem, you need to add another delay after you turn the LED off. Your code should now look like this:



```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delay(500) ;
  digitalWrite(13, LOW);
  delay(500);
}
```

Try again. Your LED should blink away merrily once per second.

## Homework - The Basics

1. All variable names in C programming are case sensitive. (true/false) \_\_\_\_\_
2. The function `digitalWrite()` takes two parameters. What is the purpose of each parameter and what type can they be?
  - a) Parameter 1: \_\_\_\_\_
  - b) Parameter 2: \_\_\_\_\_
3. The function `delay()` only takes one parameter. What is its purpose and what type can it be? \_\_\_\_\_
4. If a function has more than one parameter they must be separated by a \_\_\_\_\_.
5. What are the two **boilerplate** functions that **must** be part of every sketch?  
How does their use differentiate them one from the other?  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
6. The two boilerplate functions are preceded by the word `void`. What does this word indicate?  
\_\_\_\_\_  
\_\_\_\_\_
7. What do we call the group of C commands that start and end with curly braces?  
\_\_\_\_\_

8. What do the letters IDE stand for? \_\_\_\_\_
9. There are two **very** important buttons that we must use every time we are coding a new sketch. They are at the top of every Arduino IDE display. What are they called and what is their purpose?
  - a) Button 1 : \_\_\_\_\_
  - b) Button 2 : \_\_\_\_\_
10. What is the maximum number of **bytes** that any sketch can be? \_\_\_\_\_
11. What is a **byte**? \_\_\_\_\_
12. What are the two modes that a pin can be assigned? \_\_\_\_\_
13. What is the number of instructions that the microcontroller can perform in one second?  
\_\_\_\_\_

## Add-Ons & Projects Unit 1 - Basics

- |  |  |
|--|--|
| <b>1. Blink</b>                        | <i>handout</i>   |
| <b>2. Voltage-Ohms-Amps-Watts</b>      | <i>handout</i>   |
| <b>3. Current Construction Kit</b>     | <i>download from <a href="http://phet.colorado.edu/">http://phet.colorado.edu/</a></i> |
| <b>4. Voltage, Ohms and Resistance</b> | <i>youtube video from Bozeman Science</i>  |
| <b>5. Resister Ratings</b>             | <i>handout</i>   |

## Unit 2    Variables

In a sketch it is not uncommon to need to refer to a pin designation in more than one place. Consider the **Blink** sketch below:

```
void setup ()
{
    pinMode(13, OUTPUT) ;
}

void loop()
{
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
    delay(500);
}
```

In this example, we use pin 13 and refer to it in three places. If you decided to use a different pin, say pin 13 got damaged, then you would have to change the code in three places. Similarly, if you wanted to change the rate of blinking, controlled by the parameter in the delay function, you would have to change 500 to some other number in more than one place.

Variables can be thought of as giving a name to a number. Actually, they can be much more powerful than that, but for now, we will use them for this purpose.

When defining a variable in C, you have to specify the type of the variable. In our example we want the parameters for our variables to be whole numbers, which in C are called integers and they are indicated by the abbreviation `int`. So to define a variable called `ledPin` with a value of 13, we need to write the following:

```
int ledPin = 13;
```

Notice that because `ledPin` is a name, the same rules apply as those of function names. So, we cannot have any spaces. In addition there is a custom to start variables names with a lowercase letter and begin each new word in the variable name with an uppercase letter. Programmers will often call this "bumpy case" or "camel case." We will follow this convention throughout our course.

Let's fit this idea into the **Blink** sketch above as follows:

```
int ledPin = 13;
int delayPeriod = 500;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
  digitalWrite(ledPin, LOW);
  delay(delayPeriod);
}
```

We may have noticed that we have also made another variable called `delayPeriod`.

Now I am sure you have noticed that everywhere in the sketch where you used to refer to 13, you can now refer to `ledPin`, and everywhere you used to refer to 500, you can now refer to `delayPeriod`.

If you want to make the sketch blink faster, you can simply change the value of `delayPeriod` in one place. Try changing it to 100 and running the sketch on your Arduino board.

There are other clever things that you can do with variables. Let's modify our sketch so that the blinking starts really fast and gradually gets slower and slower, as if the Arduino is getting tired. To do this, all you need to do is to add 100 to the `delayPeriod` variable each time that you do a blink.

Modify the sketch by adding the single line at the end of the loop function so that it appears, as in the following listing, and then run the sketch on the Arduino board. If you press the **RESET** button on your board and you can

watch the blinking start from the fast rate of flashing again.

```
int ledPin = 13;
int delayPeriod = 100;

void setup ()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    delayPeriod = delayPeriod + 100;
}
```

Your Arduino is doing arithmetic now. Every time that `loop` is called, it will do the normal flash of the LED, but then it will add 100 milliseconds to the variable `delayPeriod`. We will come back to arithmetic shortly, but first we need a better way than a flashing LED to see what the Arduino is up to. We need to learn to use the Serial Monitor. That will be covered in the **Arithmetic and the Serial Monitor** unit.

## Homework - Variables

1. A variable can be thought of as giving a name to a number. (true/false) \_\_\_\_\_
2. In defining a variable in C we need to give it two things. A name and a \_\_\_\_\_.  
\_\_\_\_\_
3. What is the use of the **RESET** button found on the Arduino board and why would we ever need to use it? \_\_\_\_\_  
\_\_\_\_\_
4. With respect to variable names what does “bumpy case” or “camel case” refer to?  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
5. All variable names are case sensitive. (true/false) \_\_\_\_\_
6. Write a C command that defines a variable named `ledPin` to be of type integer and assign it the value 13. \_\_\_\_\_
7. Write a C command that defines a variable named `delayPeriod` to be of type integer and assigns it a value of 500. \_\_\_\_\_



## **Project Unit 2 - Variables**

### **1. LED Fade with POT**

*handout*

### **2. Blinking Light – Voltage Divider**

*handout*

## Unit 3      Coding Style

The compiler does not really care about how you lay out your code. For all it cares, you can write everything on a single line with semicolons between each statement. However, "well laid-out", neat code is much easier to read and maintain than poorly laid-out code. In this sense, reading code is just like reading a book: **Formatting is important**. In our class you will be expected to follow the recommendations outlined below.

Please note that the C language has a de facto coding style standard that has evolved over the years, and in this class's examples we will stay faithful to that standard.

### Indentation

In the example sketches that you have seen, you can see that we often indent the program code from the left margin. So, for example, when defining a void function, the void keyword is at the left margin, as is the opening curly brace on the next line, but then all the text within the curly braces is indented. The amount of indentation does not really matter. Some people use two spaces, some four. You can also press TAB to indent. In this class, we will always use two spaces for indentation.

Furthermore if you were to have an if statement inside a function definition, then once again you would add two more spaces for the lines within the curly braces of the if command, as in the following example:

```
void loop ()
{
    static int count = 0;
    count = count + 1;
    if(count == 20)
    {
        count = 0;
        delay(3000) ;
    }
}
```

You might want to include another `if` inside the first `if`, which would add yet another level of two space indentation, making a total of six spaces from the left margin.

All of this might sound a bit trivial, but if you ever sort through someone else's badly formatted sketch, you will find it very difficult. Then you will appreciate these simple guidelines.

## Opening Braces

There are two schools of thought as to where to put the first curly brace in a function definition, if statement, or for loop. One way is to place the curly brace on the line after the rest of the command. This is the style we have used in all the examples so far and is shown below:

```
void loop ()
{
    static int count = 0;
    count = count + 1;
    if ( count == 20)
    {
        count = 0;
        delay(3000);
    }
}
```

Alternately we could use this style:

```
void loop () {
    static int count = 0;
    count = count + 1;
    if ( count == 20)
    {
        count = 0;
        delay(3000);
    }
}
```

This style is most commonly used in the **Java** programming language, which shares much of the same syntax as C. In this class we will **always use** the first form, *placing the curly brace on the line **after** the rest of the command*, which is the form most commonly used in the Arduino world.

## Whitespace

The compiler ignores spaces, tabs and new lines, apart from using them as a way of separating the "tokens" or key words in your sketch. Thus the following example will compile without a problem:

```
void loop () {static int  
count=0;count = count + 1;  
if( count==20) {count=0;  
delay (3000);}}
```

This will work, but good luck trying to read it. Spacing, indentation, consistent formatting is critical to making maintainable code. For example:

Where assignments are made some people will write the following:

```
int a = 10;
```

But others will write the following:

```
int a=10;
```

Which of these two styles you use really does not matter, but it is a good idea to be consistent. **In this class we will always use the first form, placing a space before and after the equal sign.**

## Comments

Comments are text that is kept in your sketch along with all your real program code, **but** they actually performs *no programming function whatsoever*. The sole purpose of comments is to be a reminder to you or others as to why the code is written as it is. A comment line may also be used to present a title, date and author for a sketch.

The compiler will completely ignore any text that is marked as being a comment. I often included comments as titles at the top of many of the sketches I produce.

There are two forms of syntax for comments:

- The single line comment that starts with `//` and *finishes at the end of the line*
- The *multiline comment* that starts with `/*` and ends with a `*/`

The following is an example using both forms of comments:

```
/* A not useful loop its function is to
illustrate the concept of comments.
*/
void loop()
{
    static int count = 0;
    count = count + 1; //a single line comment
    if (count == 20)
    {
        count = 0;
        delay(3000);
    }
}
```

In our class, we will mostly stick to the single-line comment format. Good comments help explain what is happening in a sketch or how to use the sketch. They are useful if others are going to use your sketch but equally useful to yourself when you are looking at a sketch that you have not worked on for a few weeks.

Some people are told in programming courses that the more comments, the better. The best programmers will tell you that well-written code requires very little if any comments because it is self-explanatory. You should use comments for the following reasons:

- To explain anything you have done that is a little tricky or not immediately obvious
- To describe anything that the user needs to do that is not part of the program; for example, **// this pin should be connected to the transistor controlling the relay**
- To leave yourself notes; for example, **// to-do: tidy this – it's a mess**

This last point illustrates a useful technique of **to-dos** in comments. Programmers often put to-dos in their code to remind themselves of something they need to do later. They can always use the search facility in their integrated development environment (IDE) to find all occurrences of **// to-do:** in their program.

The following are *not* good examples of reasons you should use comments:

- To state the blatantly obvious; for example, `a = a + 1;` **// add 1 to a.**
- To explain badly written code. **Don't comment on it; just write it clearly in the first place.**

## Homework - Coding Style

1. What is recommended use of indentation when writing a sketch?

---

---

2. White space is made up of spaces, tabs, and new lines “\n”. How does the compiler treat “White space”.

---

---

---

3. There are two types of comments used in C programs. What are they and what is the syntax for each.

---

---

---

---

---

---

---

---

4. Name three reasons why a programmer would want to use a comment.

1) \_\_\_\_\_

2) \_\_\_\_\_

3) \_\_\_\_\_

5. Name two reasons why a programmer should not use a comment.

1) \_\_\_\_\_

2) \_\_\_\_\_

6. The following program will not cause an error when it is compiled. However it is very hard to understand. Rewrite the program so that it will follow our recommended coding style.

```
void loop () {static int count=0;count = count + 1;  
if(count==20) {count=0; delay (3000);}}
```

---

---

---

---

---

---


---

---

## Unit 4 Arithmetic and the Serial Monitor

In this unit we want to see how to perform some arithmetic operations using C programming with the Arduino hardware. But to do that we need a way to test the programs we write and see the result that our programs produce.

One way to do this is to use the Interactive Development Environment (IDE) and write the C programs then test and evaluate them on the Arduino. And then display any output result back to the *Serial Monitor*, which is a very important part of the IDE.

You access the Serial Monitor by clicking on the rightmost icon in the IDE toolbar . Its purpose is to act as a communication channel between your computer and the Arduino. You can send a message entered in the text area at the top of the Serial Monitor to the Arduino by pressing **Return** or clicking **Send**. Furthermore, if the Arduino needs to send a message to you that message will appear in the Serial Monitor screen which is below the text area at the top of the Serial Monitor. In both cases, the information is sent through the USB link between the two processors.

As you would expect, there is a built-in function that you can use in your sketches to send a message to the Serial Monitor. It is called `Serial.println()` and it expects a single argument, which consists of the information that you want to send. This information is usually a variable. Input, however, is a bit more involved. We will discuss it after we have worked with sending messages to the Serial Monitor.

We will use this Serial Monitor mechanism to test out a few things that you can do with variables and arithmetic in C; frankly, this mechanism is the only way you can see the results of arithmetic experiments in C.

### Numeric Variables, Arithmetic and the Serial Monitor

If you recall below is the sketch that makes the built in Arduino light on pin 13 blink with a steadily increasing period. The last line, highlighted below, is what causes the period to increase:

```
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
    pinMode(ledPin, OUTPUT);
}
```



```

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    delayPeriod = delayPeriod + 100;
}

```


Look closely at this line. It consists of a variable name, then an equal sign, then what is called an expression `delayPeriod + 100`. The equals sign does something called an assignment. That is, it assigns a new value to a variable, and the value it is given is determined by what comes after the equals sign and before the semicolon. In this case; the new value given to the variable `delayPeriod` is the old value of `delayPeriod` plus 100.

## Serial Output

Let's test out our serial output mechanism described above to see what the Arduino is up to. Entering the following changes below to our sketch. We added two lines. In `setup()` function we added the line `Serial.begin(9600);` and in the main `loop()` function we will added `Serial.println(delayPeriod);`.

The command `Serial.begin(9600);` instructs the IDE to create a communication channel between the Arduino and the Serial Monitor in your IDE. The 9600 refers to the rate at which the communication is to be performed. That is 9600 Baud, which means 9600 bits of information sent per second. That is a maximum of about 1200 characters per second.

The command `Serial.println(delayPeriod);` has the value of the variable `delayPeriod` printed out to the Serial Monitor followed by a line feed.

Now run our sketch and see how `delayPeriod` is changed by opening the Serial Monitor checking out what it displays. As mentioned above, you access the Serial Monitor by clicking on the *rightmost* icon in the IDE toolbar. 

```

int ledPin = 13;
int delayPeriod = 100;
void setup()

```

```

{
    Serial.begin(9600) ;
    pinMode(ledPin, OUTPUT);
}
void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    delayPeriod = delayPeriod + 100;
    Serial.println(delayPeriod) ;
}

```

The Serial Monitor should display a column of numbers that are getting successively larger and larger and the time between each displayed number should also be getting larger and larger.

If we wanted to display the numbers on one line, instead of a column, we would simply use the command **Serial.print(delayPeriod) ;**. Here the only difference is the `print` vs `println`. Note that `println` instructs the IDE to display the number and then perform a *line feed*. The use of `print` by itself will not have the line feed once the number is displayed.

Now let's try another experiment. Type in the sketch below.

```

void setup()
{
    Serial.begin(9600);
    int a = 2;
    int b = 2;
    int c = a + b;
    Serial.print("The value of c is ");
    Serial.println(c);
}
void loop()
{}

```

Note that this time we are doing all our programming in the `setup()` function. This is done since we only want our sketch to run once. And this time only one number should be displayed in the serial monitor. It should be 4, since  $2+2=4$ .

To take a slightly more complex example, the formula for converting a temperature in **degrees Centigrade into degrees Fahrenheit** we must multiply the degrees Centigrade by 9, divide by 5, and then add 32. So you could write that in a sketch like this:

```
void setup()
{
  Serial.begin(9600);
  int degC = 20;
  int degF;
  degF = degC * 9 / 5 + 32;
  Serial.print("The value of degF is ");
  Serial.println(degF);
}
void loop()
{}
```

There are a few things to notice here. First, we are again only in the `setup()` function. Secondly we are printing the words "The value of degF is " without a line feed and then the value of degF followed by a line feed. Thirdly, note the following line:

```
int degC = 20;
```

When we write such a line, we are actually doing two things: We are defining an `int` variable called `degC`, and we are assigning its initial value to be 20. Alternatively, you could separate these two things and write the following.

```
int degC;
degC = 20;
```

You must define a variable's type just once, essentially telling the compiler how much memory to reserve for it, in this case, integer needs two bytes. However, you can assign the variable a value as many times as you want. For example:

```
int degC;
```

```
degC = 20;  
degC = 30;
```

So, in the Centigrade to Fahrenheit example above, you are defining the variable `degC` and assigning it an initial value of 20, but when you define `degF`, it does not get an initial value. Its value gets assigned on the next line, according to the conversion formula, before its value is sent to the Serial Monitor for you to see.

Looking at the expression, you can see that you use the asterisk “\*” for multiplication and the slash “/” for division. The arithmetic operators +, -, \*, and / have an order of precedence and that is, multiplications and divisions are done first, going from left to right then, then additions and subtractions are done going from left to right. This is in accordance with the usual use of precedence in arithmetic. However, sometimes it makes it clearer to use parentheses in the expressions. So, for example, you could write the following:

```
degF = ((degC * 9) / 5) + 32;
```

As usual the operations in the parentheses will be done first.

The expressions that you write can be as long and complex as you need them to be, and in addition to the usual arithmetic operators, there are other less commonly used operators and a large collection of various mathematical functions that are available to you. We will learn about a few of these later.

## Serial Input

Now let’s consider serial input to the Arduino. This process involves a bit of coordination between the Arduino and our input device, the serial monitor. Consider having a conversation with a friend. You and your friend actively give each other *cues* as to when to talk and when to listen. For example, before your friend responds, you need to know that he has something to say. That is, you need to know when his reply is **available** so that you will listen. And then you will need to know when he is finished talking. You and your friend do this with ease and naturally. This process is exactly what goes on with your Arduino except with a bit more formality.

Type in the sketch below:

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

```

    }
    void loop()
    {
        if(Serial.available()>0)
        {
            int a = Serial.parseInt();
            int b = Serial.parseInt();
            if(Serial.read() == '\n')
            {
                int c = a + b;
                Serial.println(c);
            }
        }
    }
}

```

Please note that this time we have moved the bulk of our C commands into the main `loop()` function. This was done deliberately. In this example testing for input is done by placing the checking *for input* mechanism in the main `loop()`. We need to continually check if input is available. Input will come only when the input is ready. By its very nature this process must be done **asynchronously** simply because we do not know exactly when the input is ready.

I have made four lines in the sketch **bold**. We will discuss each:

```
if(Serial.available()>0)
```

This line instructs the Arduino to check **if** there is anything available to read as input from our serial monitor. The hardware checks, and if something is available the function `Serial.available()` will return a positive value greater than 0. If that is true the Block of Code inside the curly braces `{ }` will be run. In that block we first see the following two lines:

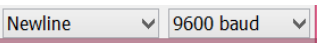
```
int a = Serial.parseInt();
int b = Serial.parseInt();
```

These lines each do two things. First they define integer variables **a** and **b** and then they assign to those variables the integer values read in from the serial monitor. The word *parse* means to “analyze the parts”. We assume that integers

have been sent from the serial monitor and the function `Serial.parseInt()` assigns those integers to the variables `a` and `b`.

The next line we encounter in our sketch is:

```
if(Serial.read() == '\n')
```

Here the Arduino is checking to see **if** the next character string read in is `'\n'`. *This is an important item.* We make an assumption that your serial monitor communication has been set in the “*new line*” mode. At the very bottom right hand side of your serial monitor screen you will see the following.  The **9600 baud** selection fits with the line `Serial.begin(9600)`; As mentioned before, it is the rate for data exchange between the Arduino and your computer over your USB cable. The **Newline** selection tells the serial monitor to add the character `'\n'` to the end of your input data whenever you hit the send button or the return key on your PC. `'\n'` is the **Newline** character!

So given the line `if(Serial.read() == '\n')` the Arduino is waiting to be sure that the send button or the return key have been pressed before it continues with its processing. Once it knows that you are done sending information the sketch adds the values `a` and `b` then assigns that value to `c`. At that point the value of `c` is printed to the serial monitor.

What you may have noticed is that all our calculations so far have been done as integers. And because of this choice the **degrees Centigrade to degrees Fahrenheit** sketch had answers that were far from accurate. We need to address this problem. We need other variable types besides integers.

## Float, a Very Important Variable Type

All our examples of variables so far have been `int` variables. This is by far the most commonly used variable type, but there are others that you should be aware of. A key type for scientific programming is `float`.

### float

The type which is relevant to calculating temperature from sensor input is `float`. This variable type represents floating point numbers that is, numbers that may have a decimal point in them, such as 1.23 or .00232. You need this variable type when whole numbers are just not precise enough.

Note the following formula:

$$f = c * 9 / 5 + 32;$$

For example if you define `f` a float variable and give `c` the value `17.0`, then `f` will be `17 * 9 / 5 + 32` or **62.6**. But if `f` is defined as an `int`, then the value will be *truncated* to **62**!

This truncation problem becomes even worse if we are not careful of the order in which we evaluate things. For instance, suppose that we did the division first, as follows:

```
f = (c / 5) * 9 + 32;
```

Then in normal math terms, the result would still be 62.6, but if all the numbers are all declared `ints`, then the calculation would proceed as follows:

- 1) 17 is divided by 5, which gives 3.4, which is then truncated to 3.
- 2) 3 is then multiplied by 9 and 32 is added to give a result of **59** which is quite a **long way** from **62.6**!

For circumstances like this, we can use floats. In the following example, our temperature conversion function is rewritten to use floats:

```
float degC = 17.0;

float degF;

degF = degC * 9.0 / 5.0 + 32.0;
```

**It is important** to notice how we have added `.0` to the end of all our constants. This ensures that the compiler knows to treat them as floats rather than `ints`. Now, as an exercise, type in the sketch below for converting a temperature in degrees Centigrade into degrees Fahrenheit using floats in place of `ints`. Input the degrees Centigrade from the serial monitor have the sketch print out the results.

```
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  if(Serial.available() > 0)
  {
    float degC = Serial.parseFloat();
    if(Serial.read() == '\n')
    {
      float degF;
      degF = degC * 9.0 / 5.0 + 32.0;
      Serial.println(degF);
    }
  }
}
```

As before I have highlighted the critical lines that perform the serial input for this sketch.

## Homework - Arithmetic & Serial Monitor

1. What is the use of the serial monitor?

---

---

---

2. What two built in functions will print the value of a variable to the serial monitor?

---

---

3. What C command would I use to print "The value of delayPeriod is: " followed by the value of the variable delayPeriod with no line feed?

---

4. Write the lines of C code that will:

- a) Define three variables `x`, `y`, and `z` to be of type `int`.
- b) Initialize `x` to 10.
- c) Initialize `y` to 20.
- d) Add the variables `x` and `y` and assign the result to `z`.
- e) Print "`x plus y is:` " followed by the value of `z`.

---

---

---

---

---



5. Write C statements to accomplish each of the following tasks:

- a) Define variables `x`, and `sum` to be of type `int`.
- b) Initialize variable `x` to 14.
- c) Initialize variable `sum` to 100.
- d) Add variable `x` to variable `sum` and assign the result to variable `sum`.
- e) Print "The sum is: " followed by the value of `sum`.

---

---

---

---

---

---

6. What **is** the *order of precedence* of the arithmetic operators `+`, `-`, `*`, and `/`?

---

Does it make a difference if you decide to use parentheses? (yes/no) \_\_\_\_\_

If it does explain how. \_\_\_\_\_

7. Write a **single** C statement that will accomplish each of the following:

- a) Increment the value of `x` by 1.
- b) Multiply the variable `x` by 2 and then assign the new value to `x`.
- c) Cube `x` and then assign the new value to `x`.

---

---

---

- d) Define `delayPeriod` to be of type integer and then assigns it a value of 500.

---

8. What does the built in function `Serial.available()` do? And how do we use it to coordinate our input from the serial monitor?

---

---

---

---

9. Write a **single** C statement to accomplish each of the following tasks:

- a) An `if` conditional statement waiting for input from the serial monitor.

---

- b) Assign to an integer `firstIn` the value received from the serial monitor.

---

- c) Assign to an integer `secondIn` the next value received from the serial monitor.

---

- d) Assign the sum `firstIn` and `secondIn` to `total`.

---

- e) Print the value of `total` to the serial monitor.

---

$m = \frac{y_2 - y_1}{x_2 - x_1}$ . Once the slope is calculated print out the slope value to the serial monitor by

[illegible]

[illegible]

[illegible]

## **Projects Unit 4 - Arithmetic & Serial Monitor**

### **1. DigitalRead() and the Serial Monitor**

*handout*

### **2. Changing Voltage with a POT**

*handout*

### **3. Servo with POT**

*handout*

## Unit 5      Control Structures

The C language has a number of built-in commands. In this section, we explore some of these and see how they can be of use in your sketches.

### Selection Structures

#### **if**   Selection Structure

In our sketches so far, we have assumed that each line of your program will be executed in order one after the other, with no exceptions. But what if you don't want to do that? What if you only want to execute part of a sketch if some condition is true?

For an example let's return to our **Gradually Slowing-Down Blinking LED** sketch from the Variables unit. In the original form it gradually get slower and slower until each blink is lasting hours. Let's look at how we can change it so that once it has slowed down to a certain point, it goes back to its fast starting speed.

To do this, you must use an **if** command structure; the modified sketch is as follows, try it out.

```
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    delayPeriod = delayPeriod + 100;
    if(delayPeriod > 3000)
    {
        delayPeriod = 100;
    }
}
```

The **if** command looks a little like a function definition, but this resemblance is only superficial. The word in the parenthesis is **not** an argument; it is what is called a *condition*. The condition is that the variable `delayPeriod` has a value that is greater than 3,000. All conditions are evaluated as Boolean value, either **true** or **false**. In this example if the condition is true, then the commands inside the curly braces will be executed and the value of `delayPeriod` is set back to 100.

If the condition is false, then the Arduino will just continue on with the next thing. In this case, there is nothing after the "if", so the Arduino will run the main loop function again.

Running through the sequence of events in your head will help you understand what is going on. So, here is what happens:

1. Arduino runs setup and initializes the LED pin to be an output.
2. Arduino starts running main loop.
3. The LED turns on.
4. A delay, of `delayPeriod` length, occurs.
5. The LED turns off.
6. A delay, of `delayPeriod` length, occurs.
7. Add 100 to the `delayPeriod`.
8. If the delay period is greater than 3,000 set it back to 100.
- 9. Go back to step 2.**

We used the symbol `<`, which means less than. It is one example of what are called *comparison operators*.

These operators are summarized in the following table:

Operator	Meaning	Examples	Result
<	Less than	5 < 6	true
		3 < 3	false
>	Greater than	6 > 5	true
		4 > 4	false
<=	Less than or equal to	5 <= 6	true
		7 <= 7	true
>=	Greater than or equal to	8 >= 4	true
		6 >= 6	true
==	Equal to	5 == 5	true
		4 == 6	false
!=	Not equal to	3 != 7	true
		2 != 2	false



To compare if two numbers are equal, you use the == command. This double equals sign is easily confused with the character =, which is used to assign values to variables.

There is another form of **if** that allows you to do one thing if the condition is true and another if it is false. We will use this in some practical examples. But let's first look at the simple example below.

## **if...else Selection Structure**

The “if...else” selection statement allows you to specify that different actions are to be performed. One set when the condition is true and another set when it is false. Consider the sketch below:

```
int ledPin = 13;
int delayPeriod = 100;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
  digitalWrite(ledPin, LOW);
  delay(delayPeriod);

  if(delayPeriod > 3000)
  {
    delayPeriod = 100;
  }
  else
  {
    delayPeriod = delayPeriod + 100;
  }
}
```

Again running through the sequence of events in your head will help you understand what is going on. So, here is what happens:

1. Arduino runs setup and initializes the LED pin to be an output.
2. Arduino starts running loop.
3. The LED turns on.
4. A delay occurs.
5. The LED turns off.
6. A delay occurs.
7. If the delay period is greater than 3,000 set it back to 100.
8. Else add 100 to the delayPeriod.
9. **Go back to step 2.**

The use of the “**if...else**” selection statement can be a wise choice when there is a choice of two action groups that depend on whether a conditional statement is true or false.

## Repetition Structures

### **for** Repetition Structure

In addition to executing different commands under different circumstances, you will also often want to run a series of commands a number of times over and over in a program. You already know one way of doing this, using the built in **main loop** function. As soon as all the commands in the main loop have been run, it will start again automatically. However, sometimes you need more control than that.

So, for example, let's say that you want to write a sketch that blinks exactly 50 times, then paused for 3 seconds, and then starts again. You could do that by just repeating the same code over and over again in your loop function, like this:

```

int ledPin = 13;
int delayPeriod = 100;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);

    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod) ;

    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);

    Now repeat the above 4 lines  
another 47 times!!!

    delay(3000);
}

```

This approach requires a lot of *copy and paste* and will result in a very large sketch. There are several *much* better ways to do this. Let's first start by looking at how you can use a **for loop** and then we will consider doing it by using a **counter** and an **if** statement.

The sketch to accomplish this with a **for loop** is below. A for loop is used when a group of C commands need to be repeated a specific number of times. We say the loop is counter controlled! As you can see it is a lot shorter and will be a lot easier to maintain than the previous example if you need to change one or more of the lines that are being repeated.

```
int ledPin = 13;

int delayPeriod = 100;


void setup()
{
  pinMode(ledPin, OUTPUT);
}


void loop()
{
  for (int i = 0; i < 50; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
  }

  delay(3000);
}
```

The **for loop**, highlighted above, looks a bit like a function that takes *three* arguments, although here those arguments are separated by semicolons rather than the usual commas. This is just a quirk of the C language. And don't worry the compiler will tell you when you get the punctuation wrong.

The first thing in the parentheses after **for** is a variable declaration. This declaration specifies the variable to be used as a counter variable and it assigns it an initial value of 0.

The second part is a *condition* that must be true for you to stay in the **for loop**. In this case, you will stay in the **for loop** as long as **i** is less than 50, but as soon as **i** is 50 or more, the program will stop doing the *Block of Code* inside the **for loop** in your curly braces { }.

The third part is what to do every time you have completed the Block of Code in the **for loop**. In this case, that is to increment **i** by **1** so that it can, after **50** trips around the **loop**, cease to be less than 50 and cause the program to exit the **loop**.

Try entering this code and running it. The only way to get familiar with the syntax and all that pesky punctuation is to type it in and have the compiler tell you when you have done something wrong. Eventually it will all start to make sense.

However there is one potential downside of this approach. Depending on what is being done the **for loop** could take a long time to run. This is *not* a problem for this sketch, because all it is doing is flashing an LED. But often, the **main loop** function in a sketch, as we have seen before, will also be checking that keys have been pressed or that serial communications have been received. If the processor is busy inside a **for loop**, it will not be able to do this. Generally, it is a good idea to make any **loop** run as fast as possible so that it can be **run** as frequently as necessary without affecting the performance of the sketch. You can often speed up a loop by exchanging the **for loop** with an **if** statement and a **counter**.

The following sketch shows how to achieve this:

```
int ledPin = 13;

int delayPeriod = 100;

int count = 0;


void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);

    count++;

    if(count == 50)
    {
        count = 0;
        delay(3000);
    }
}
```

You may have noticed the following line:

```
count ++;
```

This is just *C shorthand* for the following:

```
count = count + 1;
```

The need to use a counter that increments by 1 happens so frequently in programming that this short hand was created to make your programming task less tedious.

In the sketch above each time that loop is run, it will take just a bit more than 500 milliseconds, unless it's the 50th time round the loop, in which case it will take the same time plus the additional three seconds delay between each batch of 50 flashes.

## **While Repetition Structure**

Another way of looping in C is to use the **while** command to replace of the **for** command. You can accomplish the same task as before using a **while** command as follows:

```
int i = 0;
while(i < 50)
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);

    i++;
}
```

The expression in parentheses after **while** must be true for the program to stay in the **loop**. *In general while loops are used when a set of actions need to happen as long a certain condition remains true. It is usually used when we do **not** know in advance how many times a Block of Code will need to be repeated*, for example **while** a sensor is reading at some predetermined voltage level. However in our case we **do** know that the *Block of Code* should be repeated 50 times hence this would **not** be considered a typical use of the **while** repetition structure. At any rate when **i** is no longer less than 50, then the sketch will continues running the commands immediately after the final curly brace of our *Block of Code*.

## Homework - Control Structures

1. What is a conditional statement and what is a Boolean value?

---

---

---

2. There are six comparison operators that can be used in conditional statements. What are they and what do they mean.

---

---

---

---

---

---

3. What is the purpose of the **if...else** selection statement and why would you use it in place of the **if** selection statement.

---

---

4. In most of the **for loop** repetition structures we have seen some variate of `i++` or `count++`. What is the meaning of these C statements and why do we use them?

---

---

---

5. Write a sketch that will print out the integers from 4 to 14. Your output to the serial monitor should appear as a column of numbers. That is one line per number.

---

---

---

---

---

---

---

---

6. Write a sketch that will sum the integers from 4 to 14 and then print "The sum of the integers from 4 to 14 is: " followed by the calculated sum on the same line.

---

---

---

---

---

---

---

---



7. There is a very useful arithmetic operator “%”. It is called the Modulus Operator.

What it does is calculates the *remainder* after integer division. For example,  $(14 \% 5)$  would calculate out to be 4 and  $(17 \% 3)$  would calculate out to be 2. Using this operator and the selection statement such as `if((count % 2) == 0)` will easily indicate if the integer `count` is even or odd. Write a sketch using a `while` loop that will sum only the odd integers from 3 through 11 and then print out the sum.

---

---

---

---

---

---

8. Write a C program/sketch that will ask for an integer input and then use a `while` loop to print out the integers 1 through the integer that was your input. Your output should all be on one line.

---

---

---

---

---

---

9. Write a C program/sketch using a `while` loop that will prompt for an integer input from the serial monitor and then cube all the integers from 1 to that number. Then for each cube print “The cube of the integer “ print integer, “is: “ print the calculated cube.

[illegible]

## **Projects Unit 5 - Control Structures**

### **1. Push Down Buttons and Pullup Resistors**

*handout*

### **2. Rainbow RGB**

*handout*

### **3. Servo**

*handout*

## Unit 6 Your Own Functions & Scope

This section focuses on the type of functions that you can write yourself rather than the built-in functions such as `digitalWrite()` and `delay()` that are already defined for you. The reason that you need to be able to write your own functions is that as your sketches/code start to get complicated, your `setup()` and `loop()` functions will grow and grow until they are so long that it becomes difficult to easily understand how they work.

**The biggest problem in software development of any sort is managing complexity.** The best programmers write software that is easy to look at and understand and requires very little in the way of explanation.

Functions are the key tool in creating easy-to-understand sketches that can be changed without difficulty or change without the risk of the whole sketch falling into a crumpled mess.

### What Is a Function?

A function is a little like a program within a program. You can use a function to “Wrap up” some particular feature of your sketch that can be gathered together in one place. A function that you define can be called from anywhere in your sketch and it contains its own variables and its own list of commands. When all the commands in your function have been run, execution of your sketch will return to the point just after wherever the sketch was when the function was called.

By way of an example, code that flashes a light-emitting diode (LED) is a good example of some code that should be put in a function. So let's modify the “**blink 50 times**” sketch below to use a function that we will create called **flash**:

```
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
  pinMode(ledPin, OUTPUT);
}
```

```

void loop()
{
    for(int i = 0; i < 50; i++)
    {
        digitalWrite(ledPin, HIGH);
        delay(delayPeriod);
        digitalWrite(ledPin, LOW);
        delay(delayPeriod);
    }
    delay(3000);
}

// Blink 50 times using the function flash()

int ledPin = 13;
int delayPeriod = 250;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    for (int i = 0; i < 50; i++)
    {
        flash() ;
    }
    delay(3000);
}

void flash()
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
}

```

All we have really done here is to move the four lines of code that flash the LED from the middle of the `for()` loop to be in a function of their own called `flash()`. Now you can make the LED flash any time you like by just calling the new function by writing `flash()` in your sketch body. Note the **empty parentheses** after the function name. *This indicates that the function does not take any parameters.* The delay value that it uses is set by the `int delayPeriod = 250;` from the top of the sketch.

# Parameters

## The Key to Making Functions Versatile

When dividing your sketch up into functions, it is important to think about what service each function could provide. In the case of `flash()` above, it was obvious.

Now let's change `flash()` and give it input. That input will be in the form of parameters. Parameters that will tell our function `flash()` both, how many times to flash and how short or long the flashes should be. Read through the following code and then we will examine just how parameters work in a little more detail.

```
int ledPin = 13;
int delayPeriod = 250;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  flash(50, delayPeriod);
  delay(3000);
}

void flash(int numFlashes, int delayTime)
{
  for(int i = 0; i < numFlashes; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayTime);
    digitalWrite(ledPin, LOW);
    delay(delayTime);
  }
}
```

Now, if we look at our loop function, it has only two lines in it. We have moved the bulk of the work off to the `flash()` function. Notice how when we call `flash()` we now supply **inside** the parentheses two items called *arguments or parameters*.

Whenever we defined the function with parameters, we have to declare the type of variable we expect in our parameter list. In this case, they are both of type `int`.

**We are in fact defining new variables for our function. However, these two new variables (`numFlashes` and `delayTime`) can only be used *within* the `flash()` function. We say they are local to that function**

It is important to note that when we call the function `flash()` *both the position and type of variables used as parameters must match up with what is in the parameter list when the function is called.*

This function `flash()` would be considered a well-constructed function because it **wraps up/encapsulates** everything you need in order to flash an LED. The only additional information that it might need from outside the function is to which pin the LED is attached. If you wanted, you could also make this a parameter-*something that would be well worth doing if you had **more than one** LED attached to your Arduino.* Modify the function `flash()` to do this and then test out your new function.

## SCOPE

### Assess to a Variable: Global vs Local vs Parameters

As was mentioned before, *parameters to a function can only be used inside that function.* So, if you wrote the following code, you would get an error when you compiled it:

```
void indicate(int x)
{
    flash(x, 10);
}
x = 15;
```

On the other hand, suppose you wrote this:

```
int x;
void indicate(int x)
{
    flash(x, 10);
}
x = 15;
```

This code would *not* result in a compilation error. But in this second case you need to be careful, because you now actually have *two* variables *both called x* and they can each have different values. The one that you declared on the first line is called a *global* variable. It is called global because it can be used anywhere you like in your program, including inside any functions. We say that it can be used **globally**. The one that is passed into the function as a parameter is a local variable. We say that because it can be used only **locally** inside the function.

However, because you used the same variable name `x` inside the function, as a parameter, you cannot use the **global** variable `x` inside the function. This is simply because whenever

you refer to `x` inside the function, the "**local**" version of `x` has *priority*. The parameter `x` is said to *shadow* the global variable of the same name. This can lead to big headaches when you are trying to debug a sketch.

In addition to defining parameters that act as local variables, you can also define variables in a function itself that are not part of the parameter list. These variables are just for use within your function. These are also called **local** variables. For example:

```
void indicate(int x)
{
    int timesToFlash;
    timesToFlash = x * 2;
    flash(timesToFlash, 10);
}
```

The local variable `timesToFlash` will only exist *while the function is running*. As soon as the function has finished its last command, it will disappear. *This means that local variables are not accessible from anywhere in your program other than in the function in which they are defined.*

So, for instance, the following example will cause an error at compile time:

```
void indicate(int x)
{
    int timesToFlash;
    timesToFlash = x * 2;
    flash(timesToFlash, 10);
}

timesToFlash = 15;
```

In this case we are trying to use the `timesToFlash` variable outside the function `indicate()` and it just doesn't exist there! However, if `timesToFlash` were defined at the top of our program it would have been a global variable and there would have been no problem at compile time. This would be an easy fix but ill-advised.

***You need to know, good programmers generally treat global variables with suspicion.*** The reason is that they go against the principal of **encapsulation**. The idea of encapsulation is that you should wrap up in a package everything that has to do with a particular feature. Hence functions are great for encapsulation.

The problem with "**globals**" (as global variables are often called) is that they generally get defined at the beginning of a sketch and may then be used all over the sketch.



Sometimes there is a perfectly legitimate reason for this. Other times, people use them in a lazy way when it would be far more appropriate to pass parameters.

In our examples so far, `ledPin` is a *good* use of a global variable. It's also very convenient and easy to find up at the top of the sketch, making it easy to change. Actually, `ledPin` is really a constant, because although you may change it and then recompile your sketch, you are unlikely to allow the variable to change while the sketch is actually running. For this reason, you may prefer to use the `#define`. The `#define` command is used to associate a numerical value with a name. Everywhere the name appears in your sketch, the value will be substituted before the sketch is compiled.

Another feature of local variables is that their value is reinitialized every time the function is run. This is nowhere more true (and often inconvenient) than in the main loop function which is part of every Arduino sketch. Let's try and use a local variable in place of global variable in one of the previous examples and also use the `#define` command.

## A Curious Problem

Consider the sketch below:

```
#define ledPin 13
int delayPeriod = 250;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    int count = 0;
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
    count++;
    if(count == 50)
    {
        count = 0;
        delay(3000);
    }
}
```

In this sketch, for encapsulation sake, we want to use `count` as a local variable instead of making `count` a global variable to keep track of the number of flashes. **We want to keep `count` in the function where it is used. We want it in `loop()`.**

This sketch as written is broken. It will not work. Every time `loop()` is run, the variable `count` will be given the value 0 again, so `count` will never reach 50 and the LED will just keep flashing forever. The issue is that since `count` is only used in the main `loop()` function that is where it should be placed. However being placed there it will never serve the purpose for which it was created. It will never make it to 50. Since every time `loop()` starts up again `count` gets set back to 0 again, we have a problem. The keyword `static` to the rescue.

## Keyword `static`

Fortunately, there is a mechanism in C that gets around this problem. It is the keyword `static`. When you use the keyword `static` in front of a variable declaration in a function, it has the effect of initializing the variable *only the first time* that the function is run. That's just what is required in this situation. We can keep our variable in the function where it belongs and where it's used without it getting set back to 0 every time the function runs. The sketch below shows this in operation:

```
#define ledPin 13
int delayPeriod = 250;

void setup()
{
    pinMode(ledPin, OUTPUT);
}

void loop()
{
    static int count = 0;
    digitalWrite (ledPin, HIGH);
    delay (delayPeriod);
    digitalWrite (ledPin, LOW);
    delay (delayPeriod);
    count++;
    if (count == 50)
    {
        count = 0;
        delay(3000);
    }
}
```

# Return Values from Functions

Computer science, as an academic discipline, has as its parent's mathematics and engineering. This heritage lingers on in many of the names associated with programming. The word function is itself a mathematical term. In mathematics the input to the function (the arguments) completely determines the output. And there can only be one output for any unique set of inputs to a function. In mathematics this is the definition of a function. So far we have written functions that take only inputs, but none that have given us back a value. All our functions have been "void" functions. That is functions that do not give back any value. If a function returns a value, which it most certainly can, then you *must* specify the return value type. That type replaces the word `void`.

Let's look at writing a function that takes a temperature in degrees Centigrade and returns the equivalent in degrees Fahrenheit:

```
float centToFahren(float c)
{
    float f = c * 9.0 / 5.0 + 32.0;
    return f;
}
```

The function definition now starts with `float` rather than `void`. This indicates that the function will return a `float` to whatever calls it. A bit of code that calls it might look like this:

```
float pleasantTemp = centToFahren(20.0);
```

*All non-void functions **must** have a **return** statement in them.* If you do not put one in, the compiler will tell you that it is missing. And you can have more than one **return** in the same function. This might arise if you have an `if...else` selection structure statement with alternative actions based on some condition created by the parameters passed to the function. Some programmers frown on this but if your functions are small (**as all functions should be**), then this practice will not be a problem.

Note that the value after **return** can be an expression; it does not have to just be the name of a variable. So you could compress the preceding example into the following:

```
float centToFahren (float c)
{
    return (f = c * 9.0 / 5.0 + 32.0);
}
```

Also note that if the expression being returned is more than just a variable name, then it should be enclosed in parentheses as in the preceding example.

## More Variable Types

All our examples of variables so far have been `int` and `float` variables. These are by far the most commonly used variable types, but there are others that you should be aware of.

### boolean

Boolean values are logical. They are essentially the outcome of the evaluation of any conditional statement. That means they can only have a value that is either **true** or **false**. In the C language, Boolean is spelled with a lowercase b, but in general use, Boolean has an uppercase initial letter, as it is named after the mathematician George Boole, who invented the Boolean logic that is crucial to all computer science.

You may not realize it, but you have already met Boolean values when we were looking at the `if` command. The condition in an `if` statement, such as `(count == 50)`, is actually an expression that evaluates as a boolean result of **true** or **false**. The operator `==` is called a *comparison* operator. Whereas `+` is an *arithmetic* operator that adds two numbers together, `==` is a *comparison* operator that compares two numbers and returns a value of either **true** if the two numbers are equal or **false** if the two numbers are not equal.

You can define Boolean variables and use them as follows:

```
boolean tooBig = (x > 10);
if (tooBig)
{
    x = 5;
}
```

Boolean values can be manipulated using Boolean operators. This works in a similar way to how you can perform arithmetic on numbers. You can also perform operations on Boolean values. The most commonly used Boolean operators are **and**, which is written as `&&` and **or**, which is written as `||`.

Below you will find truth tables for the **and** “&&” and the **or** “| |” operators.

AND

		A	
		false	true
B	false	false	false
	true	false	true

OR

		A	
		false	true
B	false	false	true
	true	true	true

From the truth tables you can see that for the **and** operator, if both A and B are true then the result will be true, otherwise, the result will be false.

On the other hand, with the **or** operator; if either A or B or both A and B are true, then the result will be true. The result will be false only if neither A nor B is true.

In addition to **and** and **or**, there is the **not** operator, written as **!**. You will not be surprised to learn that "not true" is false and “not false” is true.

You can combine these operators into Boolean expressions in your `if` statements, as the following example illustrates:

```
if ((x > 10) && ( x < 50))
```

## Other Data Types

As you have seen, the `int` and occasionally the `float` data types are fine for most situations; however, some other types can be useful under some situations. In an Arduino sketch, the `int` type uses 16 bits (binary digits) or two bytes of eight binary digits each. This allows `int` to represent any number between -32767 and 32768.

Other data types are available to you to use when you program your Arduino. Those types are summarized below. This table is provided mainly for your reference. We will use some of these data types as we progress through this course.

Type	Memory Used (bytes)	Range	Notes
boolean	1	true or false (0 or 1)	
char	1	-128 to 128	Used to represent an American Standard Code for Information Interchange (ASCII) character code; e.g., A is represented as 65. Its negative numbers are normally never used.
byte	1	0 to 255	Often used for communicating serial data.
int	2	-32768 to +32767	
unsigned int	2	0 to 65536	Can be used for extra precision where negative numbers are not needed. Use with caution, as arithmetic with ints may cause unexpected results.
long	4	-2,147,483,648 to +2,147,483,647	Needed to represent only very big numbers.
unsigned long	4	0 to 4,294,967,295	See unsigned int
float	4	-3.4028235E+38 to +3.4028235E+38	double, not listed here, is same as float

An important thing to consider here is that if a data type exceeds its range, then strange and sometimes unpredictable things happen. For example if you have a `byte` variable with 255 in it and you add 1 to it, you get 0. More alarmingly, if you have an `int` variable with 32767 and you add 1 to it, you will end up with -32768.

Until you become completely comfortable with these data types, I recommend sticking to `int` and/or `float` as they will work with pretty much every time with everything.

## Homework - Your Own Functions

1. What is the biggest problem with any software development?

---

- Write a sketch that will prompt the serial monitor “Enter the number of times you want the LED on pin 13 to flash.” then get an integer input from the serial monitor and then pass that integer to the `flash()` function along with a delay of 1000 milliseconds. Observe the number of times the LED blinks. Please remember that the names of the data variables for the parameters used when calling the function `flash()` should not exactly match the names of the parameters defined in your function `flash()`.

[illegible]





4. Configure your Arduino hardware with 3 LEDs. Place them on pins 4, 5, and 6. Now rewrite the function `flash()` to take three parameters: `pinNum`, `NumFlashes`, and `delayTime`. Change your program created for problem (3) above to also prompt for the LED number. Be sure to include the LED pin choices in your prompt. Now demonstrate your experiment for your instructor.

[illegible]

5. Write a function of type `float` called `fahrenToCent` that will have one parameter `degFahr` of type `float`. This parameter will represent the degrees Fahrenheit passes into your function. Your function will return degrees centigrade.

---

---

---

---

---

6. Test your function from (5) above by writing a sketch that will prompt “Enter degrees Fahrenheit” and then take an input of a floating point variable that is passes as a parameter value to the `fahrenToCent()`. Print out “**Fahrenheit degrees of** “, print the degrees Fahrenheit here “**is equivalent to** “, print the degrees Centigrade here, “ **degrees Centigrade.**”

---

---

---

---

---

---

---

---

---

---

7. If  $x=5$  and  $y=7$  evaluate each of the following as **true** or **false**:

a)  $(x \leq 4)$  \_\_\_\_\_

b)  $(y > 7)$  \_\_\_\_\_

c)  $((x == 3) \ \&\& \ (x > 6))$  \_\_\_\_\_

d)  $((x \leq 6) \ || \ (x > 4))$  \_\_\_\_\_

e)  $((y \geq 6) \ \&\& \ (x == 5))$  \_\_\_\_\_

f)  $(( (x < 10) \ || \ (y == 7)) \ \&\& \ (x < y))$  \_\_\_\_\_

g)  $((x > y) \ || \ ((x > 3) \ \&\& \ (y == 7)))$  \_\_\_\_\_

h)  $(( (y > 10) \ || \ (x > 10)) \ || \ (x == y))$  \_\_\_\_\_

8. Using an unsigned `int` for a variable type allows your variable to take on the values from \_\_\_\_\_ to \_\_\_\_\_.

9. A boolean variable has a memory usage requirement of \_\_\_\_\_ bytes.

10. If you have the lines of code below:

```
int x = 32767;
```

```
y = x + 1;
```

then `y` must have a value of \_\_\_\_\_.

11. If you have the lines of code below:

```
byte x, y;
```

```
x = 255;
```

```
y = x + 1;
```

then `y` has the value of \_\_\_\_\_.

## **Projects Unit 6 - Functions**

- |  |                |
|--|----------------|
| <b>1. Temperature Sensor</b>                     | <i>handout</i> |
| <b>2. Push Down Buttons and Pullup Resistors</b> | <i>handout</i> |
| <b>3. Ultrasonic Sensor</b>                      | <i>handout</i> |
| <b>4. Single Servo with Blue Tooth</b>           | <i>handout</i> |
| <b>5. LED fade</b>                               | <i>handout</i> |
| <b>6. DC Motor with Transistor</b>               | <i>handout</i> |
| <b>7. Night Light</b>                            | <i>handout</i> |
| <b>8. Shift Register</b>                         | <i>handout</i> |

## Unit 7      Working with Data Structures

To write a good program, you need to think about both the algorithm (what you do) and the structure of the data you use.

You have looked at **loops**, **if** statements, and what is called the "algorithmic" side of programming an Arduino; Now we are going to turn to how you structure data. There are a large group of available data structures in the C++ computer language. The first and most fundamental of these are **Arrays**.

### Arrays

Arrays are simply a way of making an organized list of values. The variables that you have met so far have contained only a single value, usually an integer, `int`. By contrast, an array contains a list of values, and you can access any one of those values by indicating its index/position in the list/array.

First you need to know that C, in common with the majority of programming languages, begins its index/positions into an array at **0 rather than 1**. This means that the first element of every array is actually element zero.

To illustrate the use of arrays, we will create an example application that repeatedly flashes "SOS" in Morse code using the Arduino board's built-in LED. You can attach an LED anode to pin 13 with a resistor of  $220\Omega$  to the cathode and ground if you prefer.

Morse code used to be a vital method of communication in the 19th and 20th centuries, because of its coding the letters of our alphabet as a series of long and short dots. Using long and short dots for letters made it possible for messages to be sent over *telegraph wires, over a radio link, and using signaling lights*. The letters "SOS" (an acronym for "save our souls") is still recognized as an international signal of distress. The letter "S" is represented as three short flashes (dots) and the letter "O" by three long flashes (dashes). **You are going to use an array of integers to hold the *duration* of each flash that you are going to make.** You can then use a `for()` loop to step through each of the items in the array, making a flash of the appropriate duration for each item.

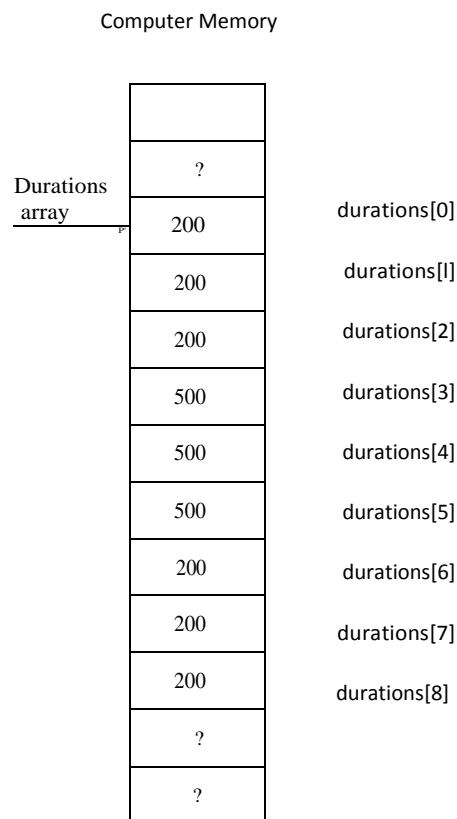
First let's take a look at how you are going to create an array of type integer containing the durations.

```
int durations[] = {200,200,200,500,500,500,200,200,200};
```

Note the syntax used here. You indicate that a variable contains an array by placing `[]` after the variable name. In this example, you are going to set the values for the durations at the time that you create the array. The syntax for doing this is to use curly braces and then values each separated by commas. Don't forget the semicolon on the end of the line.

You can access **any** given element of the array using the square bracket notation. So, if you want to get the first element of the array, you can write the following:

```
durations[0]
```



To demonstrate this idea, let's create an array and then print out all its values to the Serial Monitor. Type in the following sketch. Note the **index must be an integer**.

```

int ledPin = 13;

int durations[] =
{200,200,200,500,500,500,200,200,200};

void setup()
{
    Serial.begin(9600);
    for (int i = 0; i < 9; i++)
    {
        Serial.println(durations[i]);
    }
}

void loop()
{}

```

Upload this sketch to your Arduino board and then open the Serial Monitor. If all is well, you will see a column of numbers that match the array list you had when defining the list for the array durations.

This sketch is easily modified as needed and that makes it very useful. If you wanted to add more durations numbers to your array, all you would need to do is add them to the list inside the curly braces and then change "9" in the `for ()` loop to the new size of your array.

You have to be a little careful with arrays, because the compiler will not stop you from accessing elements of data that are beyond the end of the array. This is because the array name is really just a *pointer* to an address in memory the location of the first element in the array.

Programs keep their data, both ordinary variables and arrays, in memory. Computer memory is arranged in a very structured way. It is easiest to think of the memory in an Arduino as a collection of cubbyholes. When you define an array of nine elements the next available nine cubbyholes are reserved for its use and the variable, in this case `durations`, is said to point at the first cubbyhole or element of the array.

Going back to our point about access being allowed beyond the bounds of your array, if you were to decide to access `durations[10]`, then you would still get back an integer, but the value of this integer could be *anything*. This is in itself is fairly harmless, except that if you accidentally get a value outside of the array, you are likely to get confusing results from your sketch.

However, what is far worse is if you try changing a value **outside** of the size of the array. For instance, if you were to include something like the following

```
    durations[10] = 0;
```

The results could simply break your sketch. However the cubbyhole `durations[10]` may be in use as some completely different variable. And this could create a **serious** problem for the whole software environment that you are working in! So always make sure that you do not go outside of the size of your array. **The simple rule is that if you are using arrays and your sketch starts behaving strangely, then first check the bounds of your index/position that access your arrays.**

## Completed Morse Code SOS Using Arrays

```
#define SIZE 9
int ledPin = 13;
int durations[] = {200, 200, 200, 500, 500, 500,
200 200, 200};
void setup()
{
    pinMode(ledPin, OUTPUT);
}
void loop()
{
    for (int i = 0; i < SIZE; i++)
    {
        flash(durations[i]);
    }
    delay (1000);
}
void flash(int delayPeriod)
{
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
}
```



Remember that the obvious advantage of this approach is that it is very easy to change the Morris Code message by simply altering the `durations` array. Furthermore an integer constant, such as `SIZE` above, is often used to specify the size of an array because it will make your program more scalable (i.e. more easily expanded or contracted.)

Note that we could have initialized, as indicated below, the array `durations` to all zeros and then read in the values from the serial monitor.

Also note that in place of **`#define SIZE 9`** we *could* have used `const int size = 9;` That is:

```
const int size = 9;
int durations[size] = {0};
```

**But** by using **`#define SIZE 9`** we have *saved* the memory needed to store the value of the `SIZE` variable. As mentioned before, `#define` plays an interesting role in C programming. By using `#define` at the top of a program, *notice there is no equal sign and there is no semicolon at the end of the line*, the compiler knows to replace `SIZE` with the number `9` at the time of compiling. In other words the line **`#define size 9`** is never part of the running code and *no memory* is needed to keep track of a `SIZE` variable.

## Homework – Control Structures - Arrays

For problems 1–6 fill in the blanks for each:

1. A lists of values can be stored in a(n) \_\_\_\_\_.
2. The elements of an array are related by the fact that they have the same \_\_\_\_\_.
3. The number used to refer to a particular element of an array is called its\_\_\_\_\_.
4. When referring to an array element, the position number contained within parentheses is called a(n) \_\_\_\_\_
5. The contents of a particular element of an array is called the \_\_\_\_\_ of the element.
6. The name of the 5th element of an array `durations[]` is \_\_\_\_\_.

For problems 7-11 state whether the statement is **true** or **false**. If **false** explain.

7. An array can store many different types of values. \_\_\_\_\_  
\_\_\_\_\_
8. An array subscript can be of data type float. \_\_\_\_\_  
\_\_\_\_\_
9. To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element. \_\_\_\_\_  
\_\_\_\_\_
10. To indicate that 100 locations should be reserved for integer array `p`, we must write  
`int p[100]` \_\_\_\_\_  
\_\_\_\_\_
11. A sketch that initializes the elements of a 15-element array to zero *must* contain a  
`for()` statement. \_\_\_\_\_  
\_\_\_\_\_

For problems 12-16 write a single statement that performs each.

12. Create a constant `SIZE` and set its value to 10.

---

13. Define the array `fractions` with size 35 and elements of type float and initialize the elements to 0.

---

14. Name the forth element of the `fractions` array.

---

15. Assign the value 1.667 to `fractions` array element nine.

---

16. Assign the value 3.333 to the seventh element of the array `fractions`.

---

17. Write a part of a short sketch that will print all the elements of the array `fractions` to the serial monitor in a column format. Use a **`for()`** loop for your repetition structure.

---

---

---

---

---

---

---

---

For problems 18-21, find the errors in each program segments and then correct the error.

18. #define SIZE 100; \_\_\_\_\_

19. SIZE = 10; \_\_\_\_\_

20. #define MYVALUE = 300 \_\_\_\_\_

21. Assume:

```
int b[10] = {0}, i;
```

```
for(i = 0; i <= 10; i++) b[i] = 1;
```

\_\_\_\_\_

## **Projects Unit 7 – Working with Data Structures - Arrays**

### **1. Christmas Songs**

*handout*

### **2. Many Twinkling Lights**

*handout*

## Unit 8      Strings and Programming by Intention

In the programming world, the word string has nothing to do with long thin stuff that you tie knots in. A string is a sequence of characters. It's the way you can get your Arduino to deal with **text**. For example, the sketch below will repeatedly send the text "Hello" to the Serial Monitor one time per second:

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

void loop()
{
    Serial.println("Hello");
    delay(1000);
}
```

### String Literals

String literals are enclosed in double quotation marks. They are literal in the sense that the string is a constant, rather like the `int 123`. As you would expect, you can put strings in a variable. There is also an advanced string library, but for now you will use standard C strings, such as the one in sketch above. In C, a string literal is actually an **array of the type char**. The type `char` is a bit like `int` in that it is a number, but that number is always between 0 and 127 and represents only one character. The character may be a letter of the alphabet, a number, a punctuation mark, or a special character such as a tab or a line feed.

These number codes for letters use a standard called ASCII. Some of the most commonly used ASCII codes are shown below.

Character	ASCII code (decimal)
a-z	97-122
A-Z	65-90
0-9	48-57
space	32

The string literal “Hello” is actually an **array of characters**, as will be discussed below. And every string literal has a special *null character*, ‘\0’, at the end of each string and this null character is used to indicate the end of the string. We often check for this character to know when we have reached the end of our string.

## String Variables

As you would expect, string variables are very similar to array variables, except that there is a useful shorthand method for defining their initial value.

```
char name[] = "Hello";
```

This defines an array of characters and initializes it to the word “Hello.” It will also add a final null value ‘\0’ (ASCII 0) to mark the end of the string.

Memory

H (72)
e (101)
l (108)
l (108)
o (111)
\0 (0)

Although the preceding example is most consistent with what you know about writing arrays, it would be **more** common with literal strings to write the following:

```
char* name = "Hello";
```

This is equivalent, and the \* indicates the variable `name` is *pointer* to an array of type `char`. The idea is that `name` *points* to the first `char` element of the `char` array. That is, it points to the memory location that contains in this case the letter ‘H’.

Rewrite sketch above to use a variable as well as a string constant, as follows:

```
char name[] = "Hello";

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

void loop()
```

```

{
    Serial.println(name);
    delay(1000);
}

```

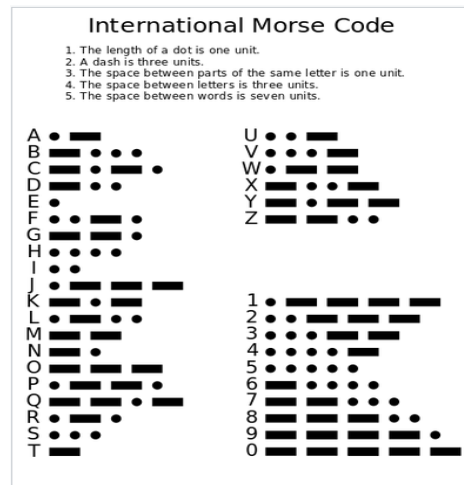
Note the first line could also have been written: `char *name = "Hello";`

Teacher sign off: \_\_\_\_\_

## “Programming by Intention”

### The Morse code Translator Example

Let’s put together what you have learned about arrays and strings to build a more complex sketch that will accept any message from the Serial Monitor and flash it out on the built-in LED.



As indicated above the rules of Morse code are that a dash is three times as long as a dot, the time between each dash or dot is equal to the duration of a dot, the wait time between two letters is the same time as a dash (three dots), and the wait time between two words is the same duration as seven dots.

For this project, we will not worry about punctuation, although there will be an exercise at the end of this unit for you to try adding punctuation to the sketch. For a full list of all the Morse code characters, see [wikipedia.org/wiki/Morse\\_code](http://wikipedia.org/wiki/Morse_code).

## Data to represent the codes

You are going to build this example a step at a time, starting with the data structure that you are going to use to represent the codes.

It is important to understand that there is no one solution to this problem. Different



programmers will come up with different ways to solve it. So, it is a mistake to think to yourself, “I would never have come up with that.” Well, no, quite possibly you would come up with something different and better. Everyone thinks in different ways, and this solution happens to be the one that first one that came to my mind.

Representing the data in this exercise is all about finding a simple way of expressing the Morris codes in C. We are going to split the data into two tables: one for the letters, and one for the numbers. The data structure for the letters is as follows:

```
char* letters[] = {
    ".-", "-...", "-.-.", "-..", ".", ".-.-", "--.", "...", "..",      // A-I
    ".---", "-.-", ".-.-.", "--", "-.", "---.", "-.-.", "-.-.", ".-.",    // J-R
    "...", "-", "...", "...-", ".--", "-.-.-", "-.-.", "-.-." };          // S-Z
```

What you have here is an *array of string literals*. So, because a string literal is actually an array of char, what you really have here is an **array of arrays**. Something that is perfectly legal and really quite useful.

For example: `letters[2]` is the pointer to the string `“-.-.”`. The `*` following the reserved word `char` indicates that the variable name `letters` is a pointer to the first element of our array of strings, that is `letters[0]`. Here we have a pointer to an array of pointers!

This means that to find the Morse code for A, you would access `letters[0]`, which would give you the string `“.-.”`. This approach is not terribly efficient, because you are using a whole byte (eight bits) of memory to represent a dash or a dot, which could be represented in a bit. However, you can easily justify this approach by saying that the total number of bytes is still only about 90 and we do have 2K memory to play with. Keep in mind that in most cases it is more important to make your code easy to understand than to worry about the amount of memory in use.

Numbers use the same approach:

```
char* numbers[] = {"-----", ".-----", "..----", "...---", "....-",      // 0-4
    ".....", "-.....", "--....", "---...", "----." };          // 5-9
```

## Globals and Setup

You need to define a couple of global variables: one for the delay period for a dot, and one to define which pin the LED is attached to:

```
int dotDelay = 200;
int ledPin = 13;
```

The setup function is pretty simple; you just need to set the `ledPin` as an output and set up the serial port:

```

void setup()
{
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);
}

```

## The *loop* function

You are now going to start on the real processing work in the loop function. The algorithm for this function is as follows:

- If there is a character to read from USB:
- If it's a letter, flash it using the letters array and wait three times the dot delay
- If it's a number, flash it using the numbers array and wait three times the dot delay
- If it's a space, wait four times the dot delay

That's all. You should not think too far ahead. This algorithm represents what you want to do, or what your intention is, and this style of programming is called *programming by intention*. If you write this algorithm in C, it will look like this:

```

void loop()
{
    char ch;
    if (Serial.available() > 0)
    {
        ch = Serial.read();
        if (ch >= 'a' && ch <= 'z')
        {
            flashSequence(letters[ch - 'a']);
        }
        else if (ch >= 'A' && ch <= 'Z')
        {
            flashSequence(letters[ch - 'A']);
        }
        else if (ch >= '0' && ch <= '9')
        {
            flashSequence(letters[ch - '0']);
        }
    }
}

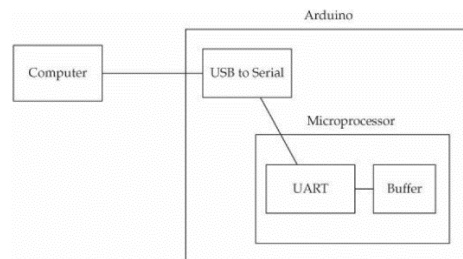
```

```

else if (ch >= ' ')
{
    delay(dotDelay * 4); // gap between words is 7 dotDelays.
}
// 3 for letter delay and this 4 is 7
}
}

```

There are a few things here that need explaining. First, there is `Serial.available()`. To understand this, you first need to know a little about how an Arduino communicates with your computer over USB. The below summarizes this process.



In the situation where the computer is sending data from the Serial Monitor to the Arduino board, the USB input is converted from the USB signal levels and protocol to something that the microcontroller on the Arduino board can use. This conversion happens in a special-purpose chip on the Arduino board. The data is then received by a part of the microcontroller called the Universal Asynchronous Receiver/Transmitter (UART). The UART places the data it receives into a buffer. The buffer is a special area of memory (128 bytes) that can hold data that is removed as soon as it is read.

This communication happens regardless of what your sketch is doing. So, even though you may be merrily flashing LEDs, data will still arrive in the buffer and sit there until you are ready to read it.

You can think of the buffer as being a bit like an e-mail inbox. The way that you check to see whether you “have mail” is to use the function **`Serial.available()`**. This function returns the number of bytes of data in the buffer that are waiting for you to read. If there are no messages waiting to be read, then the function returns 0. This is why the `if` statement checks to see that there are more than zero bytes available to read, and if they are, then the first thing that the statement does is read the *next* available char, using the function `Serial.read()`. This function’s returned value gets assigned to the local variable `ch`.

Next is another `if` to decide what kind of thing it is that you want to flash:

```
if (ch >= 'a' && ch <= 'z')
{
    flashSequence(letters[ch - 'a']);
}
```

At first, this might seem a bit strange. You are using `<=` and `>=` to compare characters. You can do that because, as noted above, each character is actually represented by a number (its **ASCII** code). So, if the code for the character is somewhere between `a` and `z` (97 and 122), then you know that the character that has come from the serial monitor is a lowercase letter. You then call a function you have not yet written, we will call it `flashSequence`, to which you will pass a string of dots and dashes; for example, to flash the letter `a`, you would pass `“.-”` as its argument. This string will come from the `letters` array.

You will be giving the responsibility to the function, `flashSequence`, for actually doing the flashing. You will not do it inside the loop. This lets us keep our code easy to read and understand.

for lower case letters here is one line of C code that determines the string of dashes and dots that you need to send to the `flashSequence` function. That line is:

```
flashSequence(letters[ch - 'a']);
```

Once again, this looks a little strange. The function appears to be subtracting one character from another. But here this is actually a perfectly reasonable thing to do, because the function is actually subtracting the ASCII values of those characters.

Remember that you are storing the Morris codes for the letters in an array whose index is an integer. Our characters when written surrounded by single quotes are actually integer values between 0 and 126. So since the first element of the array contains a string of dashes and dots for the letter `a`, the second element includes the dots and dashes for `b`, and so on all you need to do is find the right position in the array for the letter that you have just fetched from the buffer. The position for any lowercase letter will be the character code for the letter minus the character code for `a`. So, for example, `'a' - 'a'` is actually  $97 - 97 = 0$ . Similarly, `'c' - 'a'` is actually  $99 - 97 = 2$ . So, in the following statement, if `ch` is the letter `c`, then the expression inside the square brackets would evaluate to 2, and you would get element 2 from the array, which is `.-` and exactly what we need.

What this section has just described is concerned with lowercase letters. You also have to deal with uppercase letters and numbers. These are both handled in a similar manner.

## The *flashSequence* Function

We have assumed a function called `flashSequence` exists and we have made use of it, but now we need to write it.

We have planned for it to take a string containing a series of dashes and dots and to make the necessary flashes with the correct timings.

Thinking about the algorithm for doing this, you can break it into the following steps:

- Loop through each element of the string
- For each element flash that dot or dash

Using the concept of programming by intention, let's keep this function as simple as that.

The Morse codes are not the same length for all letters, so you need to loop around the string until you encounter the end marker, `\0`, as you remember this character `\0` is placed at the end of each string when it is created. You also need a counter variable, let's call it `i`, that starts at 0, the first element in array of `char`, and is incremented by one as you consider each dot and dash in the string you are processing:

```
void flashSequence(char* sequence)
{
    int i = 0;
    while (sequence[i] != '\0')
    {
        flashDotOrDash(sequence[i]);
        i++;
    }
    delay(dotDelay * 3);
}
```

```
}
```

Notice again the use of `char*`. This tells us that the parameter, `sequence`, here is an array of characters.

Again, you delegate the actual job of flashing an individual dot or dash to a new method called `flashDotOrDash`, which actually turns the LED on and off. Finally, when the program has flashed the dots and dashes, it needs to pause for three dots worth of delay. Note the helpful use of a comment.

## The *flashDotOrDash* Function

The last function in our chain of functions is the one that actually does the work of turning the LED on and off. As its argument, the function has a single character that is either a dot (.) or a dash (-).

All the function needs to do is turn the LED on and delay for the duration of a dot if it's a dot and three times the duration of a dot if it's a dash, then turn the LED off again. Finally it needs to delay for the period of a dot, to give the gap between flashes.

```
void flashDotOrDash(char dotOrDash)
{
    digitalWrite(ledPin, HIGH);
    if (dotOrDash == '.')
    {
        delay(dotDelay);
    }
    else // must be a '-'
    {
        delay(dotDelay * 3);
    }
    digitalWrite(ledPin, LOW);
    delay(dotDelay); // gap between flashes
}
```

# Putting It All Together

Putting all this together, the full listing is shown in sketch 5-05. Upload it to your Arduino board and try it out. Remember that to use it, you need to open the Serial Monitor and type some text into the area at the top and click Send. You should then see that text being flashed as Morse code.

```
int dotDelay = 200;

int ledPin = 13;

char* letters[] = {
    ".-", "-...", "-.-.", "-..", ".", ".-.-", "--.", "....", "..",      // A-I
    ".---", "-.-", "-....", "--", "-.", "---.", "-.-.", "--.-", "-.-.",  // J-R
    "...", "-", "-.-", "-...-", "--", "-...-", "-.-", "--.." };          // S-Z

char* numbers[] = {"-----", ".----", "..---", "...--", "....-",      // 0-4
    ".....", "-.....", "--....", "---..", "----." };                  // 5-9

void setup()
{
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);
}

void loop()
{
    char ch;
    if (Serial.available() > 0)
    {
        ch = Serial.read();
        if (ch >= 'a' && ch <= 'z')
        {
            flashSequence(letters[ch - 'a']);
        }
        else if (ch >= 'A' && ch <= 'Z')
        {

```

```

        flashSequence(letters[ch - 'A']);
    }
    else if (ch >= '0' && ch <= '9')
    {
        flashSequence(letters[ch - '0']);
    }
    else if (ch >= ' ')
    {
        delay(dotDelay * 4); // gap between words is 7 dotDelays, 3
    } // for letter delay and this 4 adds to 7
}

}

void flashSequence(char* sequence)
{
    int i = 0;
    while (sequence[i] != '\0')
    {
        flashDotOrDash(sequence[i]);
        i++;
    }
    delay(dotDelay * 3);
}

void flashDotOrDash(char dotOrDash)
{
    digitalWrite(ledPin, HIGH);
    if (dotOrDash == '.')
    {
        delay(dotDelay);
    }
    else // must be a '-'
    {
        delay(dotDelay * 3);
    }
    digitalWrite(ledPin, LOW);
}

```



```
    delay(dotDelay);          // gap between flashes
}
```

This sketch includes a loop function that is called automatically and repeatedly calls a `flashSequence` function that you wrote, which itself repeatedly calls a `flashDotOrDash` function that you wrote, which calls `digitalWrite` and `delay` functions that are provided by Arduino!

This is a model of how your sketches should look. Breaking things up into functions makes it much easier to get your code working and at the same time it makes it easier when you return to it after a period

## Homework – Strings and Programming by Intention

For the problems below fill in the blanks for each:

1. A string is a \_\_\_\_\_ of characters enclosed in double quotation marks.
2. A string literal is actually a(n) \_\_\_\_\_.
3. Given the line of code `char* name = "Hello";` the variable `name` is actually a \_\_\_\_\_ to the letter \_\_\_\_\_ in the string literal `"Hello"`.
4. **ASCII** refers to \_\_\_\_\_.
5. Variables of type `char` represents a \_\_\_\_\_ that is always between \_\_\_\_\_ and \_\_\_\_\_.
6. To mark the end of a string \_\_\_\_\_ is added by the compiler.
7. Global variables are usually placed at the \_\_\_\_\_ of a sketch. They are accessible to all parts of the sketch.
8. The function `Serial.available()` is of critical importance to reading characters from the USB. Describe what it does and how it is used.

---

---

---

9. Consider the line of code below:

```
int n = 'e' - 'a';
```

What will the value of `n` be? \_\_\_\_\_. Write the appropriate sketch to verify your answer. **Teacher Sign off:** \_\_\_\_\_

10. In the `flashSequence()` function we find the line:

```
while (sequence[i] != '\0')
```

What is the purpose of this line and why is `'\0'` used here?

---

---

---

11. The `flashDotOrDash()` function is the only place in the **Morris code** sketch where an LED is set HIGH or LOW. How does the code control the length of time that the LED is on and why is it appropriate in this context?

---

---

---

12. Explain what the phrase “Programming by Intention” refers to.

---

---

---

## Projects Unit 8 – Strings and Programming by Intention

### 1. Mores Code with a buzzer and an LED.

*Design & Program*

Using the code developed in this unit and the full Mores Code table below, design and implement a sketch that will read not only letters and numbers, but also a separate array, `specialSymbols[]`, that will have the 7 punctuation symbols and the equal sign that come after the number 9. Wire your bread board to not only flash an LED but also turn off and on a buzzer to indicate the appropriate Mores codes.

Teacher Sign off: \_\_\_\_\_

**Full Mores Code Table**

<b>A</b>	<b>.-</b>	<b>M</b>	<b>--</b>	<b>Y</b>	<b>-.--</b>	<b>6</b>	<b>-....</b>
<b>B</b>	<b>-...</b>	<b>N</b>	<b>-. </b>	<b>Z</b>	<b>---..</b>	<b>7</b>	<b>---...</b>
<b>C</b>	<b>-.-. </b>	<b>O</b>	<b>---</b>	<b>Ä</b>	<b>.-.- </b>	<b>8</b>	<b>---.. </b>
<b>D</b>	<b>-.. </b>	<b>P</b>	<b>.--. </b>	<b>Ö</b>	<b>---. </b>	<b>9</b>	<b>----. </b>
<b>E</b>	<b>.</b>	<b>Q</b>	<b>--.- </b>	<b>Ü</b>	<b>..-- </b>	<b>.</b>	<b>.-.-.- </b>
<b>F</b>	<b>..-. </b>	<b>R</b>	<b>.-. </b>	<b>Ch</b>	<b>----</b>	<b>,</b>	<b>--..-- </b>
<b>G</b>	<b>--. </b>	<b>S</b>	<b>...</b>	<b>0</b>	<b>-----</b>	<b>?</b>	<b>..--.. </b>
<b>H</b>	<b>.... </b>	<b>T</b>	<b>- </b>	<b>1</b>	<b>.---- </b>	<b>!</b>	<b>..-. </b>
<b>I</b>	<b>.. </b>	<b>U</b>	<b>..- </b>	<b>2</b>	<b>..--- </b>	<b>:</b>	<b>---... </b>
<b>J</b>	<b>.--- </b>	<b>V</b>	<b>...- </b>	<b>3</b>	<b>...-- </b>	<b>“</b>	<b>.-.-. </b>
<b>K</b>	<b>-. - </b>	<b>W</b>	<b>.-- </b>	<b>4</b>	<b>....- </b>	<b>‘</b>	<b>.----. </b>
<b>L</b>	<b>.-.. </b>	<b>X</b>	<b>-.- - </b>	<b>5</b>	<b>..... </b>	<b>=</b>	<b>-...- </b>

## Unit 9      Object Oriented C++

In the C++ programming language the “++” tells us that our C programming language has Object orientation. Object orientation uses a concept called **classes** to aid in **encapsulation**. *Encapsulation* is one of the four fundamentals of object-oriented programming. *Encapsulation* refers to the bundling of data with the methods (functions) that operate on that data into small pieces of code called classes. It is a critical part of all large project programming. Generally, a **class** is like a section of a program that includes both variables—called *member variables*—and *methods*, which are like functions but apply to the **class**. These functions can either be public, in which case these methods/functions may be used by other classes, or private, in which case the methods can be called only by other methods within the that class.

Whereas most of our Arduino sketches have been contained in a single file, when you are working in C++, and in particular with the new Photon hardware you tend to use more than one file. In fact, there are generally at least **two files** for every class: A *header* file, which has the extension **.h**, h stands for header, and the *implementation* or C++ file, which has the extension **.cpp**, cpp stands for C++..

### Built-in Library Example

For example in working with a servo we would need to use a preexisting library. Our program would start with the include command to include the file Servo.h:

```
#include <Servo.h>
```

This file is the header file for the **class** called **Servo**. This file tells the Arduino sketch what it needs to know to be able to use the library. You can actually retrieve this file if you go to your Arduino installation folder and file and find the file libraries/Servo. You *will* need to open the file in a text

editor. If you are using a Mac, then right-click on the Arduino app itself and select the menu option Show Package Contents. Then navigate to Contents/Resources/Java/libraries/Servo. If you are using Windows, then right-click on the Arduino app and select the menu option Open File Location, then navigate to Arduino/libraries/Servo/src, *src* stands for “SouRCe”, then open Servo in a text editor, I prefer Wordpad.

The file Servo.h contains lots of code, as this is a fairly large library class. The code for the actual class itself, where the nuts and bolts of running a servo actually reside, are in the file Servo.cpp.

In the next section, you will create a simple example library that should put the concepts behind a **library** into context.

## Libraries

Creating an Arduino library might seem like the kind of thing that only a seasoned Arduino veteran should attempt, but actually it is pretty straight-forward to make a library. For our example, we will convert into a library the *flash* function from **Unit 6**. As you might recall the flash function caused an LED to flash for a specified number of times.

To create the C++ files that are needed to do this, you will need a text editor for your computer—something like WordPad on Windows or Text-Mate on Mac. Start by creating a folder to contain all these library files. You should create this folder inside the libraries folder of your Arduino documents folder. In Windows, your libraries folder will be in OS(C:)/ProgramFiles(x86)/Arduino/libraries. On the Mac, you will find it in your home directory, Documents/Arduino. If there is no libraries folder in your Arduino directory, then create one.

This libraries folder is where any libraries you write yourself, or any “unofficial” contributed libraries, must be installed.

Now we will create a folder within our **Arduino/library** folder to hold our new library. Call the folder will be called **Flasher**. If you are not sure where your Arduino folder is located simply right click on

the Arduino app icon and choose Arduino and then properties and finally OpenFileLocation. Now start the text editor you prefer and type the following into it:

```
// LED Flashing library
#include "Arduino.h"
class Flasher
{
public:
    Flasher(int pin, int duration);
    void flash(int times);
private:
    int _pin;
    int _d;
};
```

Save this file in the Flasher folder with the name Flasher.h. This is the **header file** for this library class. This file specifies the different parts of the class. As you can see, it is divided into public and private parts.

The public part contains what looks like the start of two functions. These are called methods and differ from functions only insofar as they are associated with a class. They can be used only as part of the class. Unlike functions, they cannot be used on their own.

**The first method, *Flasher*, begins with an uppercase letter, which is something you would not use with a function name. It also has the same name as the class. This *Flasher* method is called a **constructor**, which you can use to create a new **Flasher object** to use in a sketch.**

For example, you could put the following in a sketch:

```
Flasher slowFlasher(13, 500);
```

This would create a **new Flasher object** called `slowFlasher` that would flash on pin D13 with a duration of 500 milliseconds.

The second method in the class is called `flash`. This method takes a single argument of the number of times to flash. Because it is associated with a **class**, when you want to call it, you have to refer to the object that you created earlier, as follows:

```
slowFlasher.flash(10);
```

This would cause the LED to flash ten times at the period that you specified in the **constructor** to the Flasher object. The private section of the class contains two variable definitions: one for the pin, and one for the duration, which is simply called `_d`. Every time that you create an object of class Flasher, *it will have these two variables*. This enables any object you create of the class Flasher to remember its pin and duration so that when you `slowFlasher.flash()` it the new object knows where and how long to do it.

These variables, `_d` and `_pin`, are called member variables because they are members of the same **class**. Their names generally are unusual in that they start with an underscore character. However another commonly used naming convention is to use a lowercase m (for member) as the first letter of these variable names.

## The Implementation File \*.cpp

The header file has just defined *what the class looks like*. You now need a separate file that *actually does the work*. This is called the implementation file and has the extension **.cpp** (C++).

So, create a new file containing the following and save it as *Flasher.cpp* in your Flasher folder:

```
#include "Arduino.h"
#include "Flasher.h"
Flasher::Flasher(int pin, int duration)
{
    pinMode(pin, OUTPUT);
    _pin = pin;
    _d = duration / 2;
}
void Flasher::flash(int times)
{
    for (int i = 0; i < times; i++)
    {
        digitalWrite(_pin, HIGH);
        delay(_d);
        digitalWrite(_pin, LOW);
        delay(_d);
    }
}
```

There is some unfamiliar syntax in this file. The method names, Flasher and flash, are both prefixed by **Flasher::**. This indicates that these methods belong to the Flasher class.



The constructor method (Flasher) just assigns each of its parameters to the appropriate private member variable. The duration parameter is divided by two before being assigned to the member variable `_d`. This is because the delay is called twice, and it seems more logical for the duration to be the total duration of the flash and the gap between flashes.

The flash function actually carries out the business of flashing; it loops for the appropriate number of times, turning the LED on and off for the appropriate delay.

## Completing Your Library

You have now seen all of the essentials for your library. You could now deploy this library and it would work just fine. However, there are two further steps that you should take to complete your library. One is to define the **keywords** used in the library so that the **Arduino IDE** can show them in the **appropriate color** when users are editing code. The other is to include some examples of how to use the library.

## Keywords

To define the keywords, you have to create a file called **keywords.txt**, which goes into the **Flasher directory**. This file contains just the two following lines:

```
Flasher      KEYWORD1
flash        KEYWORD2
```

This is essentially a two-column table in a text file. The left column is the keyword and the right column an indication of the type of keyword it is. **Class names** should be a **KEYWORD1** and **methods** should be **KEYWORD2**. It does not matter how many spaces or tabs you put between the columns, but each keyword should start on a new line.

## Examples

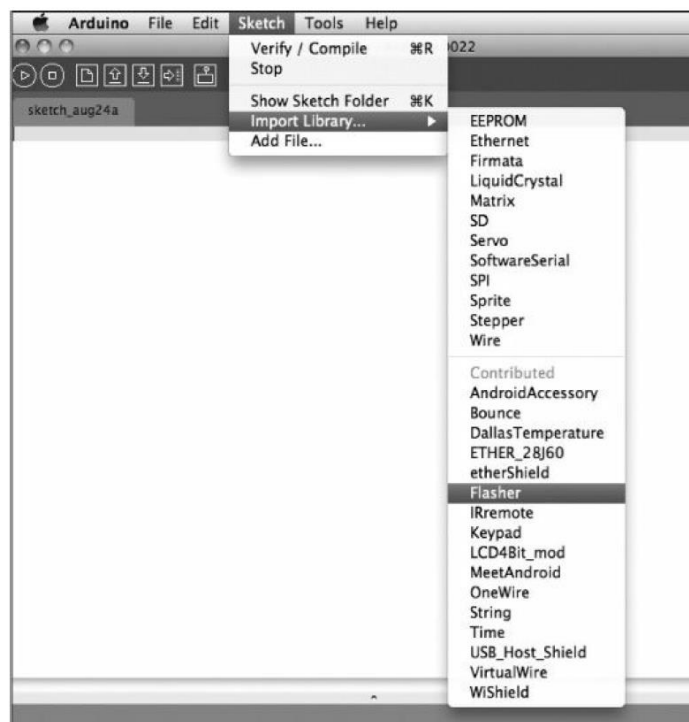
The other thing that you, as a good Arduino citizen, should include as part of the library is a folder of examples. In this case, the library is so simple that a single example will suffice.

The examples must all be placed in a **folder called examples** inside the **Flasher** folder. *The example is in fact just an Arduino sketch*, so you can create the example using the Arduino IDE. But first, you have to *quit and then reopen* the Arduino IDE to make it aware of your new library.

After restarting the Arduino IDE, from the Arduino IDE's menu, select **File** and then **New** to create a new sketch window. Then from the *Menu Bar*, select **Sketch** and then **Include Library** option. The Options should look something like the figure below.

The libraries above the line in the submenu are the official libraries; below this line are the “unofficial” contributed libraries. If all has gone well, you should see Flasher in the list.

If Flasher is not in the list, it is very likely that the Flasher folder is not in the libraries folder of your sketches folder, so go back and check.



Type the following into the sketch window that has just been created:

```
#include <Flasher.h>

int ledPin = 13;

int slowDuration = 300;

int fastDuration = 100;

Flasher slowFlasher(ledPin, slowDuration);

Flasher fastFlasher(ledPin, fastDuration);

void setup() {}

void loop()

{

    slowFlasher.flash(5);

    delay(1000);

    fastFlasher.flash(10);

    delay(2000);

}
```

The *Arduino IDE* will not allow you to save this example sketch directly into the *libraries* folder, so save it somewhere else under the name **Simple Flasher Example** and then move the whole Simple Flasher Example that you just saved into the *examples* folder in your Arduino folder.

If you restart your Arduino IDE, you should now see that you are able to open the example sketch from the menu as shown in the figure below.

## Conclusion

There is much more to C++ and to writing libraries, but this unit should get you started. It should also be sufficient for most of what you are likely to do with an Arduino. These Arduinos are small devices

and the temptation is often to over engineer solutions that could otherwise be very simple and straightforward.

That concludes the main body of this workbook. For further information on Arduino and where to go next, a good starting point is always the official Arduino website at [www.arduino.cc](http://www.arduino.cc).

If you are looking for help or advice, the Arduino community on **[www.arduino.com/forum](http://www.arduino.com/forum)** is extremely helpful.

## Homework – Object Oriented C++

Fill in the blanks for each:

1. What does the “++” tell us about the C++ programming language? \_\_\_\_\_.
2. Encapsulation is a fundamental of object oriented programming. What is encapsulation and why is it so important for large programming projects?

---

---

---

3. Use the internet search to find the other three fundamentals of object-oriented programming. List each below and include a short explanation of what each is about.

---

---

---

---

---

---

---

4. When referring to an array element, the position number contained within parentheses is called a(n) \_\_\_\_\_.
5. The contents of a particular element of an array is called the \_\_\_\_\_ of the element.
6. The name of the 5th element of an array `durations[]` is \_\_\_\_\_.

For problems 7-11 state whether the statement is **true** or **false**. If **false** explain.

7. An array can store many different types of values. \_\_\_\_\_  
\_\_\_\_\_
8. An array subscript can be of data type float. \_\_\_\_\_  
\_\_\_\_\_
9. To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element. \_\_\_\_\_  
\_\_\_\_\_
10. To indicate that 100 locations should be reserved for integer array `p`, we must write  
`int p[100]` \_\_\_\_\_  
\_\_\_\_\_
11. A sketch that initializes the elements of a 15-element array to zero *must* contain a  
`for ()` statement. \_\_\_\_\_  
\_\_\_\_\_

For problems 12-16 write a single statement that performs each.

12. Create a constant `SIZE` and set its value to 10.  
\_\_\_\_\_
13. Define the array `fractions` with size 35 and elements of type float and initialize the elements to 0.  
\_\_\_\_\_
14. Name the forth element of the `fractions` array.  
\_\_\_\_\_
15. Assign the value 1.667 to `fractions` array element nine.  
\_\_\_\_\_

16. Assign the value 3.333 to the seventh element of the array `fractions`.

---

17. Write a part of a short sketch that will print all the elements of the array `fractions` to the serial monitor in a column format. Use a **`for()`** loop for your repetition structure.

---

---

---

---

---

---

---

For problems 18-21, find the errors in each program segments and then correct the error.

18. `#define SIZE 100;` 

---

19. `SIZE = 10;` 

---

20. `#define MYVALUE = 300` 

---

21. Assume:

```
int b[10] = {0}, i;
```

```
for(i = 0; i <= 10; i++) b[i] = 1;
```

---





## Projects Unit 9 – Object Oriented C++

### 1. Build a Flasher Object.

*Program*

Write a sketch that will use the Flasher object class to create another `Flasher` object called `fancyFlasher`. This object will flash a red LED on the redPin at a duration of only 50 milliseconds. Wire your bread board appropriately.

**Teacher Sign off:** \_\_\_\_\_

### 2. Build an Object Library **Buzzer**

*Design and Program*

Design and program an object library called `Buzzer`. This library will allow the user to create objects to simplify the use of a piezo buzzer. Each object in this class will have three private data elements, `Pin`, `Frequency` and `Duration` and two functions `buzzOn` and `buzzOff`. These functions will use the Arduino library functions `tone()` and `noTone()` described below. Once the library is created program a sketch to use it that will create 4 objects called `highLong`, `lowLong`, `highShort` and `lowShort`. Any *high* object will have a frequency of 5000Hz and any *low* object will have a frequency of 1000Hz. *Long* and *short* refer to the duration of the buzz. *Long* is 3 seconds and *short* is 1/2 second.

Write a sketch to use your four objects in an interesting and creative way. Wire your bread board accordingly.

**Teacher Sign off:** \_\_\_\_\_

**Syntax for `tone()`**  
`tone(pin, frequency)`  
**Parameters**  
pin: the pin on which to generate the tone  
frequency: the frequency of the tone in hertz - *unsigned int*

**Syntax for `noTone()`**  
`noTone(pin)`  
**Parameters**  
pin: the pin on which to stop generating the tone

## Unit 10      Pointers in C++

In this section we will explore the difference between a pointer to a memory location and the value stored in that location.

Consider the line of C code:     `int a = 5;`

With this one line of code we are doing **two** things, first we are reserving a place in memory to store an integer value for the variable `a` and secondly we are storing the value `5` in that memory location.

It is important to realize that both of these things are happening each time we initialize a variable and assign a value to it.

Because there are two things happening it is not surprising that we are able to find/use both the value of the variable `a` and value of its memory location. And consequently we are able to use them in any reasonable way including passing both these values to functions via a parameter list and manipulate them via assignment statements where appropriate. We do this with the help of two special characters `'*'` and `'&'`. Study the examples below.

Consider the following Arduino sketch:

```
void setup()
{
    Serial.begin(9600);
    int a = 5;
    changeInt(a);
    Serial.println(a);
}

void changeInt(int c)
{
    c = 200;
}

void loop()
{ }
```

If you run this sketch `5` will print out on your serial monitor. If we consider the scope of the variables `a` and `c` we know this will happen since the variable `c` is a **local** variable to the function `changeInt()`. Changing `c` in `changeInt()` will not affect the value of `a` in the function `setup()` from which `changeInt()` is called.

Now consider the following Arduino sketch:

```
void setup()
{
    Serial.begin(9600);
    int a = 5;
    changeInt(&a);
    Serial.println(a);
}

void changeInt(int* c)
{
    *c = 200;
}

void loop()
{}
```

If you run this sketch 200 will print out on your serial monitor. Note the differences in the two sketches:

1. `changeInt(a)` is modified to `changeInt(&a)`

In `changeInt(a)` we are passing the value that was assigned to `a` in the line `int a = 5;`

In `changeInt(&a)` we are passing the **address** of the memory location for `a`. The `&` before the variable indicates that we will be sending an address as the parameter for `ChangeInt()`.

2. `void changInt(int c)` is modified to `void changeInt(int* c)`

In `changeInt(int c)` the function `changeInt()` is expecting to receive an integer value to assigned to `c`, in this case 5.

In `changeInt(int* c)` the function `changeInt()` is expecting to receive the address of a location in memory, in this case the memory location of the variable `a`. The `int*` indicates that the parameter variable `c` will receive a location in memory that holds an integer. In this case that location is the address of the integer `a`. In general the use of `*` in this way in a parameter list indicates that we are dealing with a **pointer** to a location in memory of the type of variable indicated.

3. `c = 200;` is changed to `*c = 200;`

`c = 200;` is a simple assignment statement. Here `c` is assigned the value of 200.

Now consider `*c = 200;`. Here the character `"*"` is added before the pointer variable `c`. `"*"` is called a **dereference** or **indirection** operator. It is always denoted by `"*"` (i.e. an asterisk) and is a unary operator that operates on *pointer variables*. In this case the location of variable `a` is passed via the parameter list to `c`. This process of assigning a value using `*c=` is called **"dereferencing"** the pointer. It accesses the **address location** `c` is *pointing to/referencing* and then, in this case, changes the value at that location to 200.

Another Arduino sketch using pointers and dereferencing:

```
void setup()
{
  Serial.begin(9600);
  int x;
  int* p;
  x = 0;
  p = &x;
  *p = 1;
  Serial.println(x);
}
```

```
void loop()
{}
```

This sketch will print 1 to the serial monitor.

Now let's consider each line of code in `setup()` above:

1. `Serial.begin(9600);` Starts up the serial monitor.
2. `int x;` Here we are declaring the variable `x` as an integer.
3. `int* p;` `"*"` is used in this declaration statement to indicate that `p` is a **pointer** to an integer in memory. And because of **dereferencing** `"*p"` must be an integer.
4. `x = 0;` This assignment statement makes the value of `x` equal 0.
5. `p = &x;` `"&x"` is the **address** for the variable `x`. In this case `p`, a pointer, is assigned the **address** of the variable `x`.
6. `*p = 1;` This is equivalent to the statement `x = 1;`. Note that `"*p"` is the **value** `p` is pointing to. Hence `*p = x`. the logic is simple, `*p = 1` and `*p = x`, so `x = 1`.
7. `Serial.println(x);` This line prints the new value of `x`, (i.e. 1), to the serial monitor.

**Summary:** `x` is an **integer** and `p` is a **pointer**. `"&x"` is the **address** of `x`'s integer value and `"*p"` is the **integer value** that `p` is **pointing to**.

## Homework – Pointers in C++

For problems 1–6 fill in the blanks for each:

1. A lists of values can be stored in a(n) \_\_\_\_\_.
2. The elements of an array are related by the fact that they have the same \_\_\_\_\_.
3. The number used to refer to a particular element of an array is called its \_\_\_\_\_.
4. When referring to an array element, the position number contained within parentheses is called a(n) \_\_\_\_\_.
5. The contents of a particular element of an array is called the \_\_\_\_\_ of the element.
6. The name of the 5th element of an array `durations[]` is \_\_\_\_\_.

For problems 7-11 state whether the statement is **true** or **false**. If **false** explain.

7. An array can store many different types of values. \_\_\_\_\_  
\_\_\_\_\_
8. An array subscript can be of data type float. \_\_\_\_\_  
\_\_\_\_\_
9. To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element. \_\_\_\_\_  
\_\_\_\_\_
10. To indicate that 100 locations should be reserved for integer array `p`, we must write  
`int p[100]` \_\_\_\_\_  
\_\_\_\_\_
11. A sketch that initializes the elements of a 15-element array to zero *must* contain a  
`for()` statement. \_\_\_\_\_  
\_\_\_\_\_

For problems 12-16 write a single statement that performs each.

12. Create a constant `SIZE` and set its value to 10.

---

13. Define the array `fractions` with size 35 and elements of type float and initialize the elements to 0.

---

14. Name the forth element of the `fractions` array.

---

15. Assign the value 1.667 to `fractions` array element nine.

---

16. Assign the value 3.333 to the seventh element of the array `fractions`.

---

17. Write a part of a short sketch that will print all the elements of the array `fractions` to the serial monitor in a column format. Use a **`for()`** loop for your repetition structure.

---

---

---

---

---

---

---

---

For problems 18-21, find the errors in each program segments and then correct the error.

18. #define SIZE 100; \_\_\_\_\_

19. SIZE = 10; \_\_\_\_\_

20. #define MYVALUE = 300 \_\_\_\_\_

21. Assume:

```
int b[10] = {0}, i;
```

```
for(i = 0; i <= 10; i++) b[i] = 1;
```

\_\_\_\_\_

## **Projects Unit 10 – Pointers in C++**

### **1. Christmas Songs**

*handout*

### **2. Many Twinkling Lights**

*handout*



