RetroLCD.com

Absolute Beginner’s Guide to Arduino

Contents

Introduction

The document is based on the book “Absolute Beginner’s Guide to C” available at a very low cost used from Amazon, and the “Arduino Reference Sheet” available from RetroLCD.com

Microsoft currently has Visual Studio Community Edition free for non-commercial use. When coding you will want to get comfortable with an IDE which will greatly help with your productivity as you continue in your learning. Good IDE’s have code competition, auto code formatting, and generally help with the more mundane aspects of programming. They also tell you about many simple mistakes before you try to compile and help you track down errors after compiling. They will also let you quickly trace your code, so you can start at the error and quickly figure out the flow of code that got to that point.

Note that the Arduino IDE does not qualify as a “good” IDE, but it is, unfortunately, the least terrible free option currently. For small programs, the lack of basic IDE functions is less terrible than it is normally. There is also Eclipse but the Arduino plugin is non-free, and Eclipse is quite clunky.

Visual Micro is a Visual Studio Plugin that looks promising but is $65 for commercial use and is sold per machine rather than per user. Non-commercial and students can install on up to 3 machines and it costs $45. There is also an annual renewal fee to keep up with the latest version.

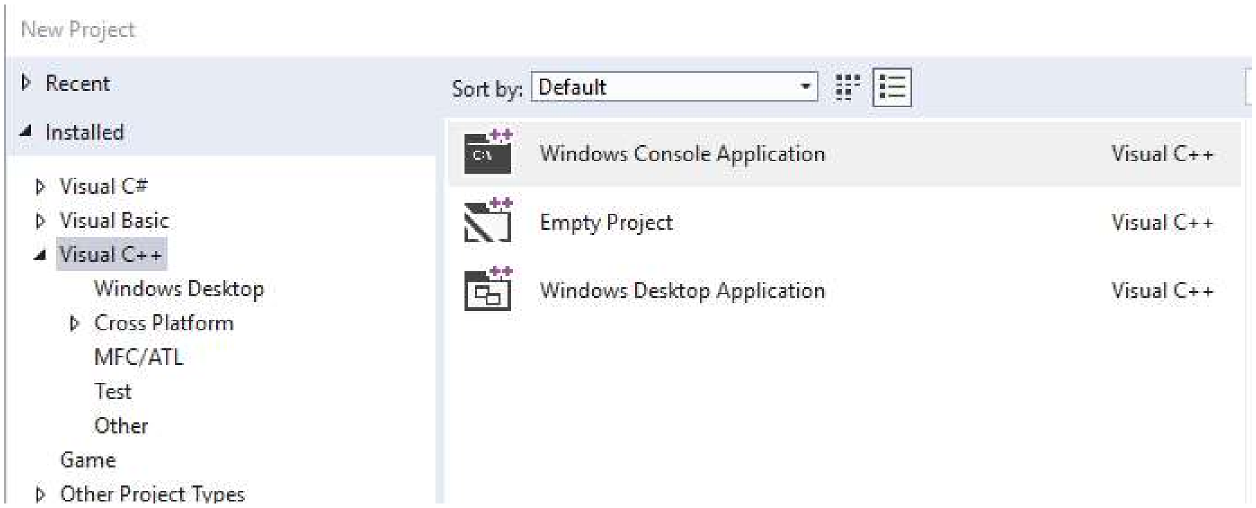
This guide will not go into significant detail because there is nothing duller than reading hundreds of pages of what is essentially a dictionary.

This guide is simply a summary to refer to and become familiar with the most fundamental keywords you will be using. They will be covered in context as we start to work through game programming.

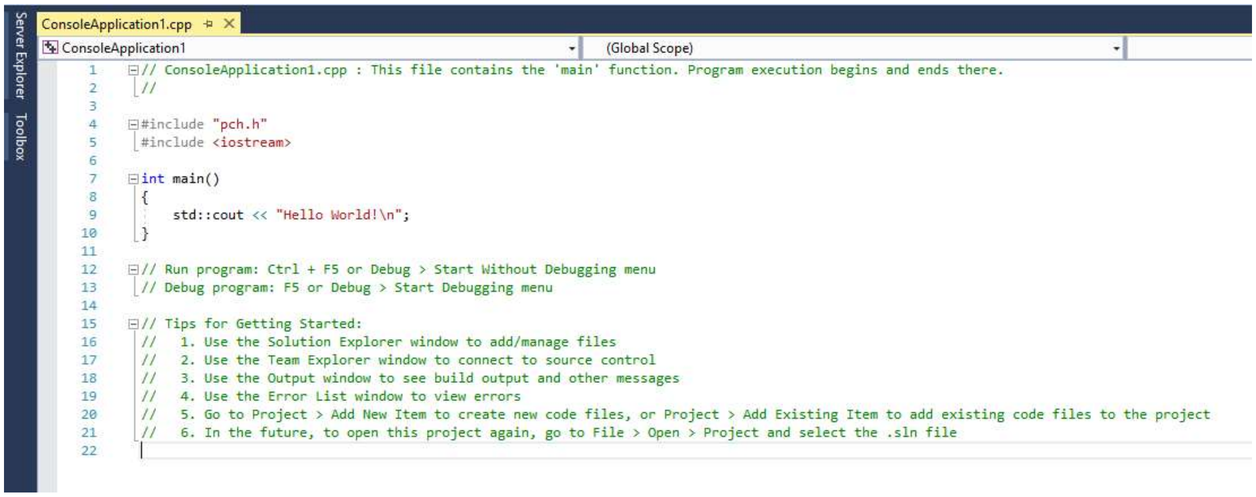
Chapter 1: Hello World

Main, Setup, Loop

In Visual Studio, go to File -> New -> Project

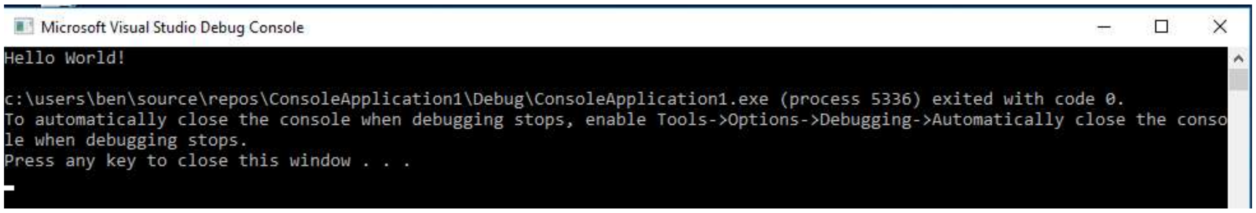


We'll be starting with a **Windows Console Application**. By default, the following program will be created for you.



```
// ConsoleApplication1.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include "pch.h"
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln file

```

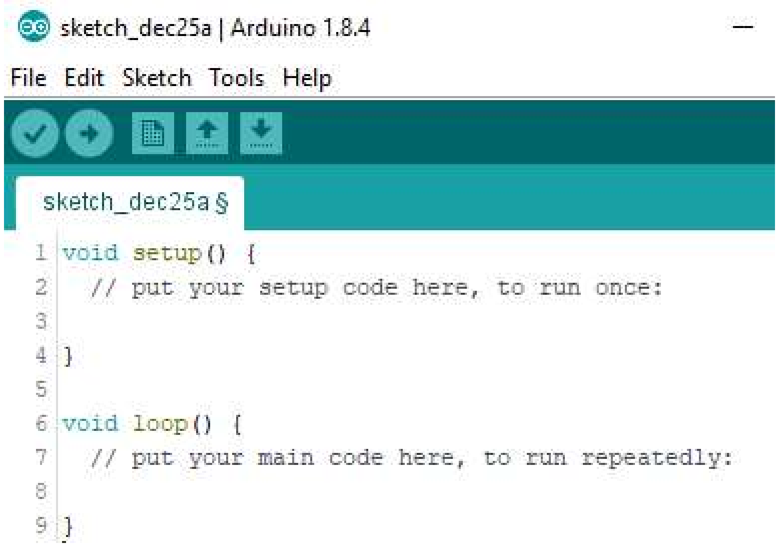
When you run the program as given you will see



```
Hello World!

c:\users\ben\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (process 5336) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

"Press any key to close this window…" is something added by **Visual Studio** so that you can see the output of the program rather than it just closing automatically. The program wants to immediately exit because there is no loop. The "main" function is called once and only once.

In the land of **Arduino** when you click File -> New you will see something like



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

}

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

}
```

Comments

Notice that in both **Visual Studio** and **Arduino** there are lines that start with //

Those are **comments**. They allow you to put text into your program that is ignored by the **compiler** but is useful for telling you, or someone else reading your code, what the code is supposed to do.

There is a whole Wikipedia article dedicated to explaining what comments are:

https://en.wikipedia.org/wiki/Comment_(computer_programming)

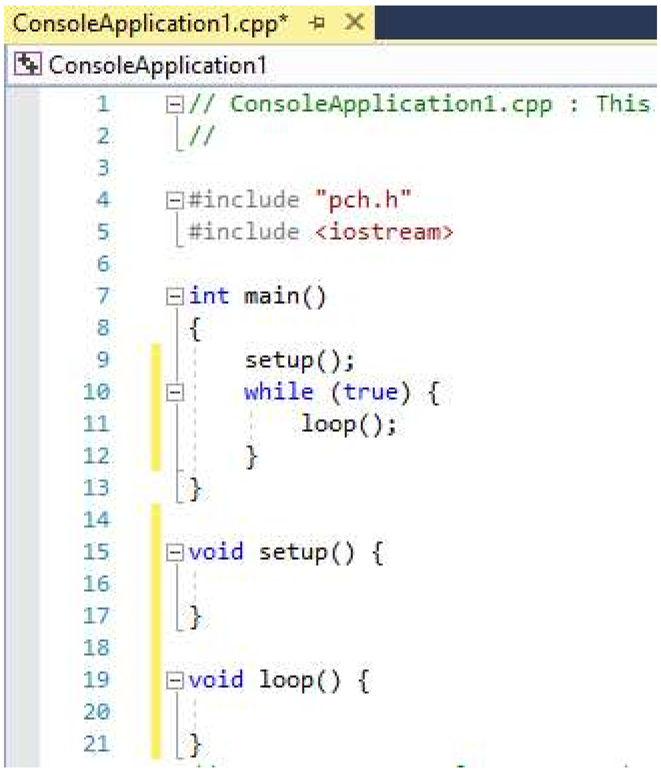
When you first starting programming, you may be tempted to comment everything because you are telling yourself how everything works. Which is fine. But, as your skills improve, your commenting should focus more on WHY your code is doing something and not HOW it is doing it. Programming is about literally learning another language. You would no more explain a line of code than you would explain a sentence in a book. The words are understood. They're on the page. Comments are only needed if what is on the page does not express the motivation. Why did you pick these particular words and put them in these particular sentences?

Note the above Arudino code. The function is "setup" and the comment says, "put your setup code here, to run once."

There are two things to notice here. The first is that the comment repeats the name of the function. The name of the function should clearly indicate what the function does. The value of the comment is that is "runs once." That is not obvious from the name of the function that it is run only once. It is very much like the "main" function in the Visual Studio example.

The next given function is called "loop" and the comment is that you should put your main code there and it will run repeatedly. Didn't we just say our "setup" function is like the "main" function?

Let's clear this up by making it explicit what the Arduino code is doing using the Visual Studio IDE.



```
// ConsoleApplication1.cpp : This
//

#include "pch.h"
#include <iostream>

int main()
{
    setup();
    while (true) {
        loop();
    }
}

void setup() {

}

void loop() {

}
```

The **Arduino IDE** is hiding the "main" function from the developer because it is reserved. Meaning, there is very specific code that must be in it for your program to run on the Arduino and rather than bother you the developer with this, setup and loop are exposed instead so you can get right to work writing your program and not deal with the lower level particulars of the Arduino.

The actual main.cpp in the **Arduino Core Library** does not match exactly what is given but is basically the same idea.

There are two other things to notice at this time:

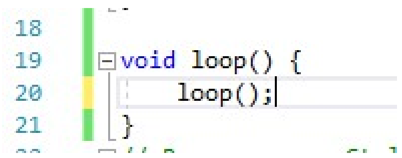
1. Blocks of code have { } around them
2. Statements such as loop(); end with a semicolon

Where the brackets go can depend largely on the IDE you use as it will format the code for you. It's important to understand the difference between syntax and function. Certain languages confuse the two and force the programmer to format the code a certain way or it will not compile.

Statements that tell the computer to do something end with a semicolon just like sentences end with a period. Some languages do not always require a semicolon. Which would be like not always requiring a period at the end of a sentence. As you advance in your programming skills you will learn that things which are technically optional just annoy the people who come later which often will be yourself. Being sloppy will generally annoy your future self more than anyone.

Recursion

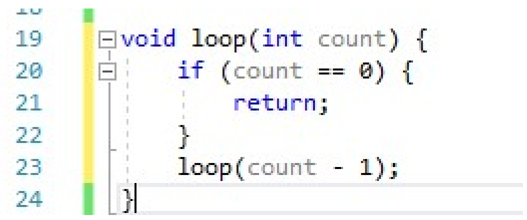
As a beginner programmer, never do this:



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

Never call a function from within itself in order to create an **infinite loop**. When creating an infinite loop, as the above main function does, a function must return before it can be called again.

Recursion is an advanced topic that we will not delve into here, but a simple valid example is as follows:



```
void loop(int count) {
    if (count == 0) {
        return;
    }
    loop(count - 1);
}
```

A recursive function must meet the following criteria:

1. Have a stopping condition
2. Have a finite number of executions.

Now, just never use it. When you are ready to use it, you will be able to explain why you need it and what the fatal flaw in the above example is.

What Have We Learned?

1. Comments are used to tell the reader WHY a section of code is doing something
2. The function setup() is called once
3. The function loop() is run an infinite number of times
4. As a beginner, a function must always return before you can call it again
5. Recursion is not a solution to your problems

Chapter 2: Numbers, Strings and Arrays Oh My

In any program it is necessary to store information. There are only three fundamental **data types** in C

- char
- int
- float

Everything else is built on top of these types. Some may consider **bool** (or **boolean**) a primary data type but it is just a special purpose **byte** that is restricted to being 1 or 0.

There are three primary considerations when looking at what type of variable you want to use:

1. What kind of information does this variable represent?
2. How large of a value do I need it to hold?
3. Can the value be negative?

Of the primary data types, only **float** is a specially encoded value. The rest are just some number of bytes. When a number can be negative, the highest bit is used to tell the processor whether the number is negative or positive. This necessarily cuts the maximum value in half. When working with a limited system like the Arduino, it is advantageous to solve your problems using unsigned chars. Meaning, every value you need to store should be between 0 and 255.

An **array** is simply a collection of a fundamental data type. An array cannot contain more than one data type. A string for example, is an array of char variables that should always have zero as the last value of the string. I you want to put the string "hello world" into a program, you would need 5 bytes for "hello," 1 byte for the space, and 5 bytes for "world" but then also a byte for zero which tells the computer that is the end of the string. At first glance you might try to store "hello world" in 11 bytes of memory but that could cause problems later. You will need 12 bytes with the 12th byte being set to zero.

What is a Byte?

For more in-depth information, check out

https://en.wikipedia.org/wiki/Byte

A byte is a collection of 8 bits. It is the smallest unit of memory available. It is not possible to have a variable which accesses only a single bit of memory. A bit can only have a value of 1 or 0. Digital circuits only understand on or off.

Memory

Memory on the Arduino is measured in kilobytes.

A kilobyte is 1024 bytes.

There are three types of memory on the Arduino:

1. Flash memory (32 kilobytes) – where your program is stored
2. SRAM (2 kilobytes) – where your variables are stored
3. EEPROM (1 kilobyte) – where your saved data is stored

It is possible to store your variables in program (flash) memory

https://www.arduino.cc/reference/en/language/variables/utilities/progmem/

Hopefully it's obvious why we need to care about the size of variables now.

To give you some perspective, the Atari 2600 had 128 bytes of memory. The original NES has 2 kilobytes of memory, the same as the Arduino UNO.

Processing Speed

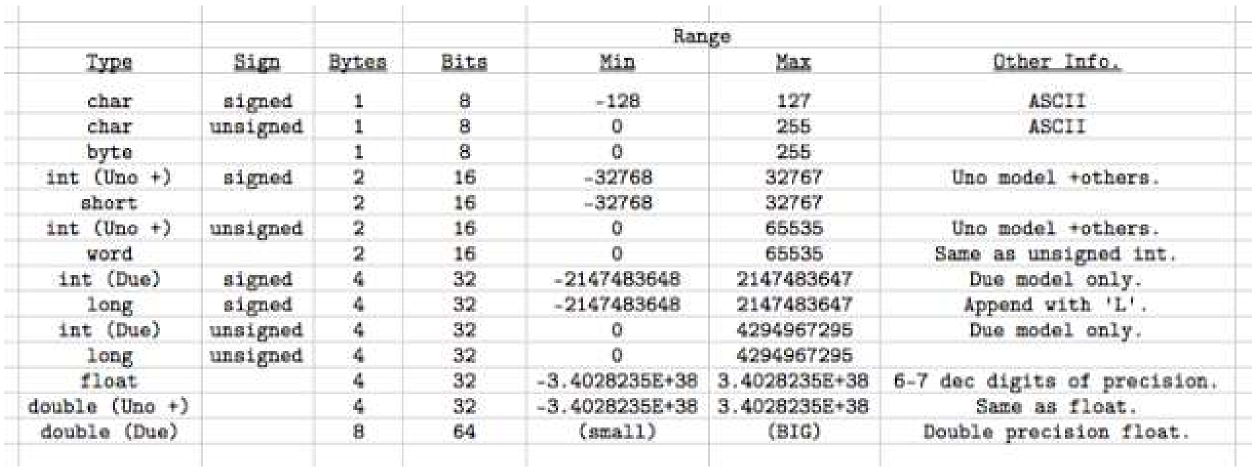
The NES ran at just under 2mhz compared to the Arduino running at 16mhz.

You might be tempted to think that because the Arduino has the same amount of memory as a Nintendo and runs at a much faster speed, you could run NES games with an Arduino.

You'd be partially correct. The Arduino would in theory run the game logic of an NES game just fine. Presumably the modern Arduino chip does math as well as or better than the Ricoh 2A03 found in the original NES. Both processors are 8-bit which means they can only process 8 bits of data in a single instruction. If you attempt to do 16-bit math on an 8-bit processor it will take at least twice as long.

The difference between an NES and an Arduino is that the NES is a collection of processors while the Arduino is a single processor. Hopefully this prompts you to start thinking about how we can make the Arduino part of a collection of processors rather than trying to make it do all the work by itself.

Thinking About Data



				Range		
Type	Sign	Bytes	Bits	Min	Max	Other Info.
char	signed	1	8	-128	127	ASCII
char	unsigned	1	8	0	255	ASCII
byte		1	8	0	255	
int (Uno +)	signed	2	16	-32768	32767	Uno model +others.
short		2	16	-32768	32767	
int (Uno +)	unsigned	2	16	0	65535	Uno model +others.
word		2	16	0	65535	Same as unsigned int.
int (Due)	signed	4	32	-2147483648	2147483647	Due model only.
long	signed	4	32	-2147483648	2147483647	Append with 'L'.
int (Due)	unsigned	4	32	0	4294967295	Due model only.
long	unsigned	4	32	0	4294967295	
float		4	32	-3.4028235E+38	3.4028235E+38	6-7 dec digits of precision.
double (Uno +)		4	32	-3.4028235E+38	3.4028235E+38	Same as float.
double (Due)		8	64	(small)	(BIG)	Double precision float.

Source: https://thmuses.wordpress.com/2014/01/05/arduino-variable-types/

The above is a list of the data types available on the Arduino, the minimum and maximum values and the number of bytes they take up.

When working with a memory restricted system there are two benefits to sticking with the minimum sized data types:

1. Maximizing processing speed
2. Maximizing available memory

Let's pretend we have a system that can hold 16 bytes of memory and we want to have one hero and 4 enemies on the screen.

Each of those entities require two variables: x and y in order to position them on the play field. So, in total we have 10 variables. And they must fit in 16 bytes.

If we were to use an int, it would take 2 bytes each which is 20 bytes which is too much. So, we must use bytes which means we have to store the position between the values of 0-255. This memory limitation then drives our game design as our levels must allow for the player to stay within those positions. If the exit to the level is at position 300, 200 and our player can only get to 255, 200, they will never be able to reach the exit.

When designing a game, it is essential to think about memory and design within the limitations of the system. It is interesting to learn that all the math to create modern 3D games was figured out decades ago. Long before it was possible for computers to use that math to create games with. The Super FX chip came about because of a couple teenagers who were bound and determined to get 3D on everything.

https://en.wikipedia.org/wiki/Argonaut_Games

They were creating 3D games on systems that were not designed with 3D in mind like the Commodore 64. Eventually, Nintendo noticed their work and decided to fund the creation of a chip that would speed up the math they were using to make it possible to run 3D games on the Super Nintendo. Their most well-known game was Star Fox. What they essentially did was create a graphics card that was part of the game cartridge rather than being built into the system.

You can always spend more money and get a faster system. The challenge of the Arduino is creating entertaining software within the limitations of the system with perhaps, some supporting hardware.

As we get into specific games, we will talk more about data types and what supporting hardware we will be producing.

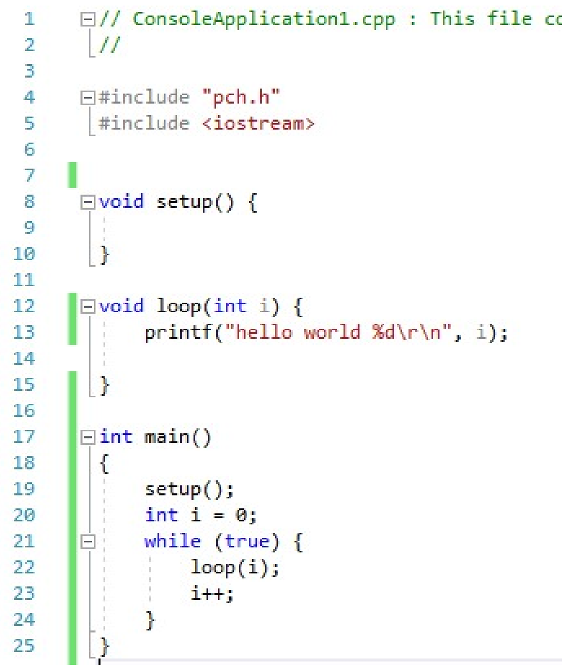
What Have We Learned?

1. The Arduino has 3 fundamental data types
2. The Arduino has 3 different memory stores
3. The Arduino has very little memory to work with
4. The Arduino requires additional hardware to create games

Chapter 3: Printing Strings

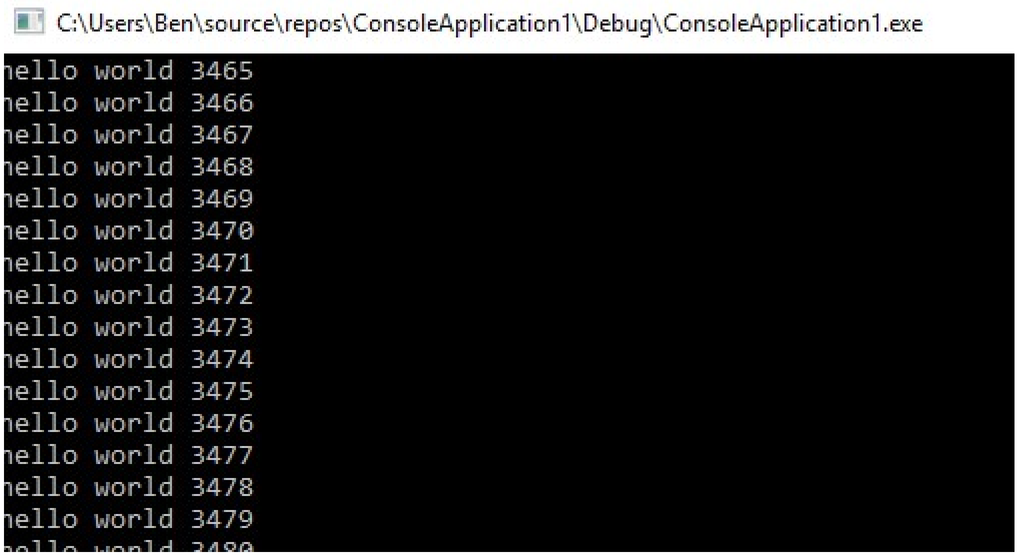
In the world of Visual Studio, we have access to a function called **printf** which is used to format and display information with relative ease.

Note: we've moved setup and loop to be defined before main so that it compiles. You cannot reference a function before it is declared.



```
// ConsoleApplication1.cpp : This file cc
//

#include "pch.h"
#include <iostream>


void setup() {

}

void loop(int i) {
    printf("hello world %d\r\n", i);

}

int main()
{
    setup();
    int i = 0;
    while (true) {
        loop(i);
        i++;
    }
}
```

This program endlessly loops and prints out something like:



C:\Users\Ben\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe

```
hello world 3465
hello world 3466
hello world 3467
hello world 3468
hello world 3469
hello world 3470
hello world 3471
hello world 3472
hello world 3473
hello world 3474
hello world 3475
hello world 3476
hello world 3477
hello world 3478
hello world 3479
hello world 3480
```

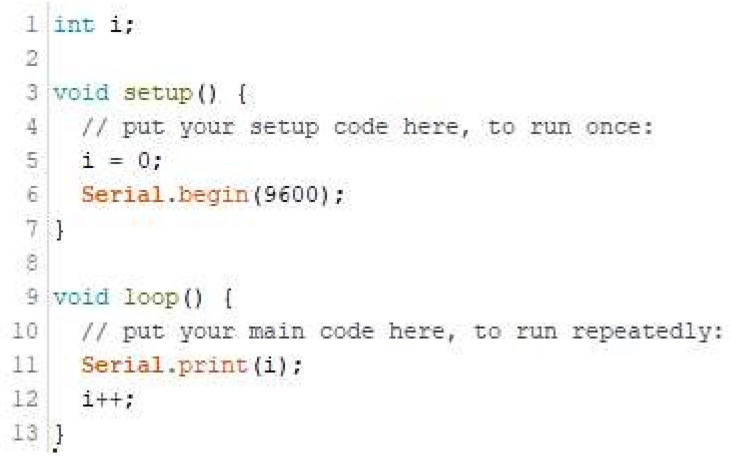
Until you close it.

The %d in the string tells the compiler that we're going to be outputting an integer. \r\n are special characters that move the cursor to the next line (\n) and to return the cursor to the beginning of the line (\r).

http://www.cplusplus.com/reference/cstdio/printf/

The above link will go into all the details you could want about how to print out different variable types.

In the land of Arduino, there is no screen.



```
int i;

void setup() {
  // put your setup code here, to run once:
  i = 0;
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print(i);
  i++;
}
```

Note: Don't forget to put Serial.begin(9600) in your setup function if you intend to send data out the serial connection. 9600 is the baud rate. You can increase or decrease it but 9600 works well enough.

Instead we have what is called the Serial Monitor.

Tools -> Serial Monitor

We use Serial.print to output some variable to the serial monitor and we use Serial.println to output some variable to the serial monitor and go to the next line. It simply adds the "\r\n" automatically.

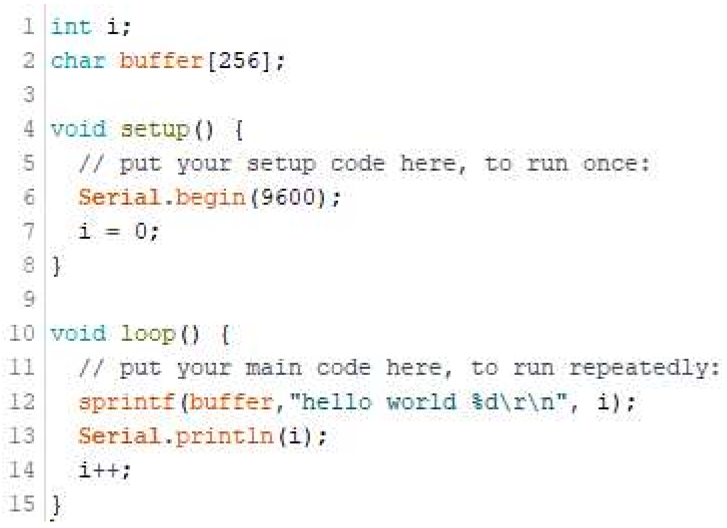
The limitation of Serial.print is that it can only print one variable at a time. With printf we were able to combine some text with a variable.

We can do the same thing with the Arduino using the sprintf function (which is also available in Visual Studio land).

http://www.cplusplus.com/reference/cstdio/sprintf/

The limitation of the sprintf function for the Arduino is that it does not support floating point values.

sprintf works the same as printf (using the same formatting codes) except that the result is put into a buffer rather than output directly to the screen.



```
int i;
char buffer[256];

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  i = 0;
}

void loop() {
  // put your main code here, to run repeatedly:
  sprintf(buffer,"hello world %d\r\n", i);
  Serial.println(i);
  i++;
}
```

Note: Don't forget to put Serial.begin(9600) in your setup function if you intend to send data out the serial connection. 9600 is the baud rate. You can increase or decrease it but 9600 works well enough.

The buffer is simply an array of char data types that must be greater than or equal to the resulting size of the sprintf function.

The return value of the sprintf function is the number of characters written to the buffer.

However, "this count does not include the additional null-character automatically appended at the end of the string." (cplusplus.com)

All strings must end with a null character. Null is simply the integer value zero. This tells the processor to stop reading bytes.

Later on, when we cover the LCD Character display, this zero is null concept makes it a curious decision to make the zero character customizable as in normal strings zero is never rendered as a character; it's empty.

What Did We Learn

1. The Arduino cannot print floating point values using the sprintf function
2. Serial.print can print anything including floating point values
3. The serial monitor is used to allow the Arduino to show the developer information

Chapter 4: Math

The Arduino supports your basic math functions that you would find on any basic calculator

- + (add)
- - (subtract)
- * (multiply)
- / (divide)
- % (remainder)
- = (assignment)

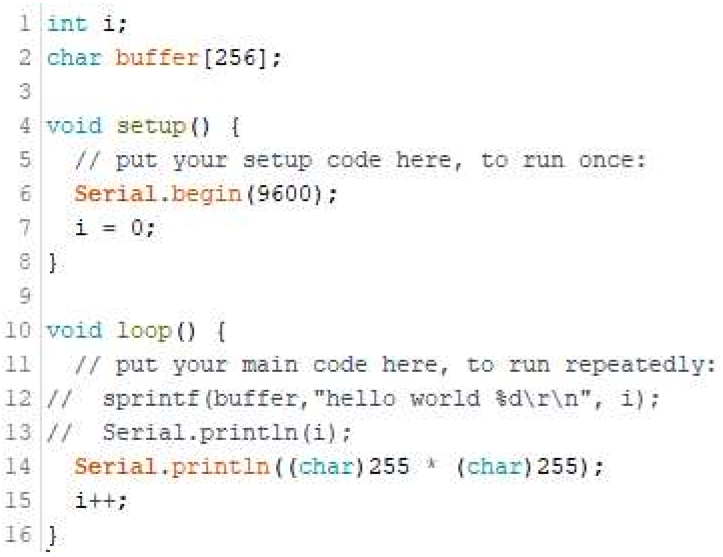
If you divide by zero your program will return either -1 or 1.

http://forum.arduino.cc/index.php?topic=42391.0

This has to do with the implementation of division. Modern computers will throw an error and halt the program, but an embedded system doesn't have time for that. If you want something to happen in the case of divide by zero, you will have to implement your own error checking.

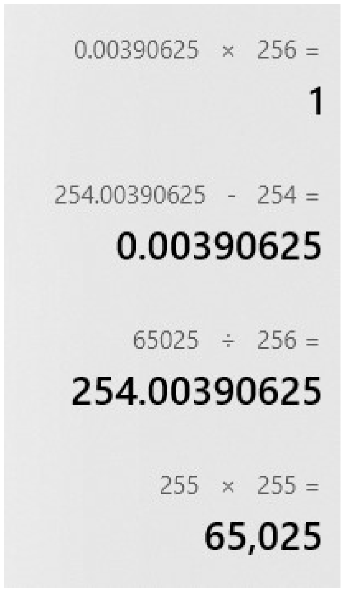
On the other end of math, you can overflow your data type.

A char can only hold a value from 0 to 255. So, if we multiply 255 * 255 we get 1.



```
int i;
char buffer[256];

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  i = 0;
}

void loop() {
  // put your main code here, to run repeatedly:
//  sprintf(buffer,"hello world %d\r\n", i);
//  Serial.println(i);
  Serial.println((char)255 * (char)255);
  i++;
}
```

Note that we have (char) before our number. This tells the compiler to use the char data type. Without that, it will treat the numbers as 16-bit (2 byte) integers instead.



```
0.00390625  ×  256 =
1

254.00390625  -  254 =
0.00390625

65025  ÷  256 =
254.00390625

255  ×  255 =
65,025
```

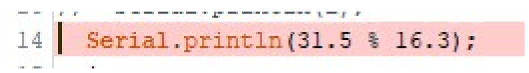
From bottom to top, this shows the math how of 255 x 255 is 1. You could also use the % operator to get the same answer.

Note: these steps demonstrate how you can easily modulus any base, even floats. See if you can write a function that can be passed two floats and return the modulus of those numbers as a float.

When dividing integers, the fractional part is ignored. There is no rounding. 31 / 16 is 1. You will not get 2 until you do 32 / 16.

Modulus / Remainder

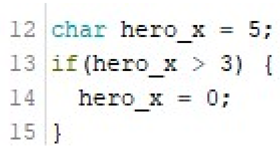
The modulus (remainder) operator is an essential part of game programming. It allows us to wrap around arbitrary values. The % operator only works on whole, integer values. Where integer referrers to the mathematical definition, not the processor definition of 16-bit vs 8-bit. You cannot do something like:



```
Serial.println(31.5 % 16.3);
```

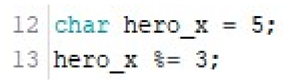
The program will not compile.

Let’s say we’re making a game and we want the hero to show up on the left side of the screen if they go off the right side of the screen. We could do something like this:



```
char hero_x = 5;
if(hero_x > 3) {
  hero_x = 0;
}
```

It would be easier to simply do:



```
char hero_x = 5;
hero_x %= 3;
```

The advantage of using an if statement to check is that you are given an opportunity to apply other game logic. Perhaps the hero can go off the screen and wrap to the other side, but they lose points. Also notice that in the first example, hero_x will go to zero. But in the second example, hero_x will go to 2.

Carefully consider the choices of your math or you may end up with unintended consequences in your program.

Chapter 5: Comparison Operators

- != (not equal to)
- < (less than)
- <= (less than or equal to)
- > (greater than)
- >= (greater than or equal to)
- == (equal to)

```
char hero_x = 5;
if(hero_x = 3) {
  hero_x = 0;
}
```

One of the more common bugs and sources of nefarious hacks is the = vs ==. = assigns a value to a variable. == compares two variables.

```
byte hero_x = 5;
if(hero_x = 3) {
  hero_x = 0;
}
 Serial.println(hero_x);
```

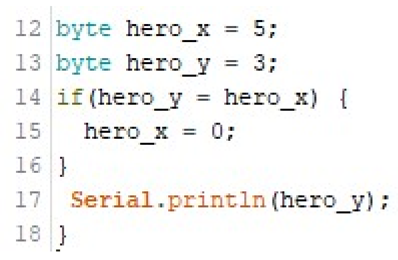
Notice that we switched to a byte. When hero_x is a char, nothing but a character return is printed when Serial.println is used because char is interpreted as a character rather than a number. When we use a byte, the number zero is printed. This is an important concept we will go into much more depth on later. Despite the same value being stored in memory, how they are interpreted is different.

One of the very newbie ways to avoid this issue is to write code like:

```
byte hero_x = 5;
if(3 = hero_x) {
  hero_x = 0;
}
 Serial.println(hero_x);
```

This results in a compiler error if you have a single = symbol. However, it makes the code far less readable. Don’t do that. Part of being a skilled developer is moving beyond dumb mistakes or at least catching them before they go out into the wild. It is like putting training wheels on your bike.

Let’s consider the following example:



```
byte hero_x = 5;
byte hero_y = 3;
if(hero_y = hero_x) {
  hero_x = 0;
}
 Serial.println(hero_y);
}
```

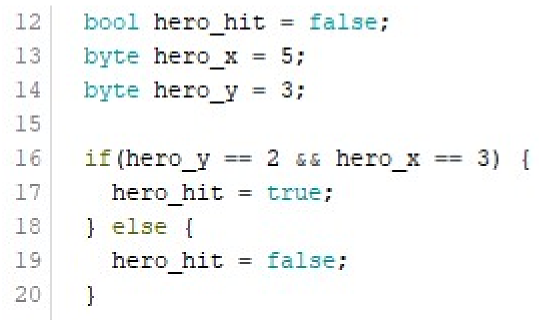
It is rare that you are checking against a specific value. Far more often you are comparing two variables as illustrated above. hero_y is now 5 and hero_x is now zero. If you were to compare hero_x to hero_y using = then hero_y would remain 3 and hero_x would be zero. Order of operations matter with assignment. They do not matter with ==.

In short, if you are expecting the compiler to save you from using = instead of ==, you are going to screw up a lot when the compiler isn't going to save you.

These kinds of mistakes say a lot about a developer and there is just no way around it. Much of programming is attention to detail.

Chapter 6: Boolean Operators

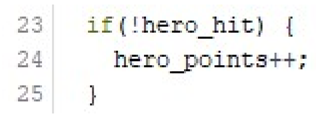
- ! (not)
- && (and)
- || (or)



```
bool hero_hit = false;
byte hero_x = 5;
byte hero_y = 3;

if(hero_y == 2 && hero_x == 3) {
  hero_hit = true;
} else {
  hero_hit = false;
}
```

Boolean operators allow us to compare multiple things.

The not operator allows us to check the inverse of a value.

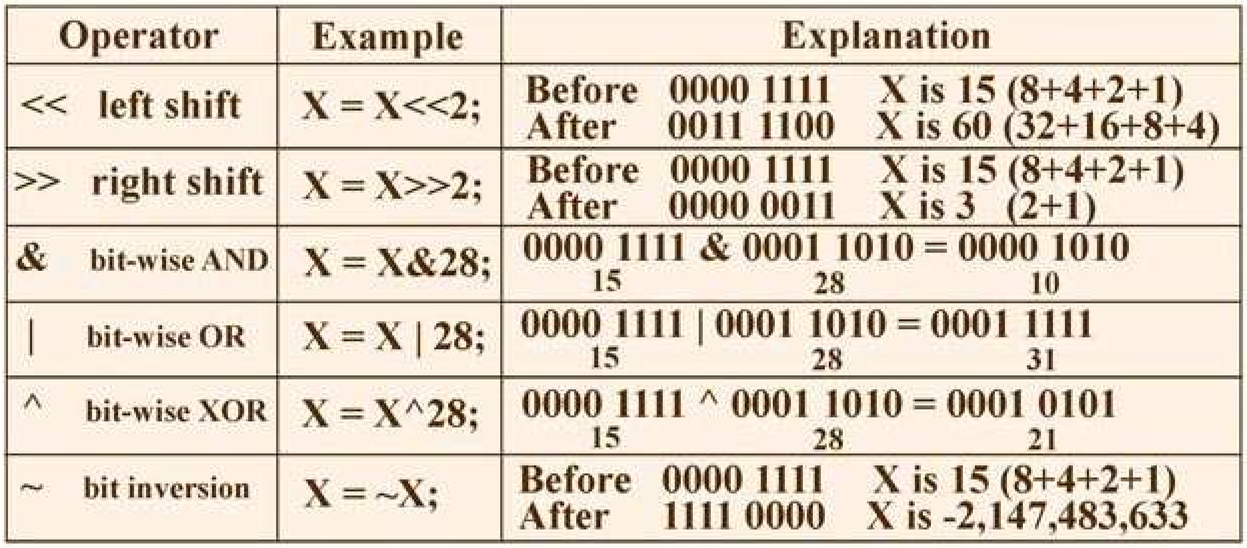


```
if(!hero_hit) {
  hero_points++;
}
```

This is where naming variables is important. If the variable to decide if the hero is hit or not was named hero_not_hit then our if statement would look different. Generally, “not” should not appear in a variable name.

Chapter 7: Bitwise Operators

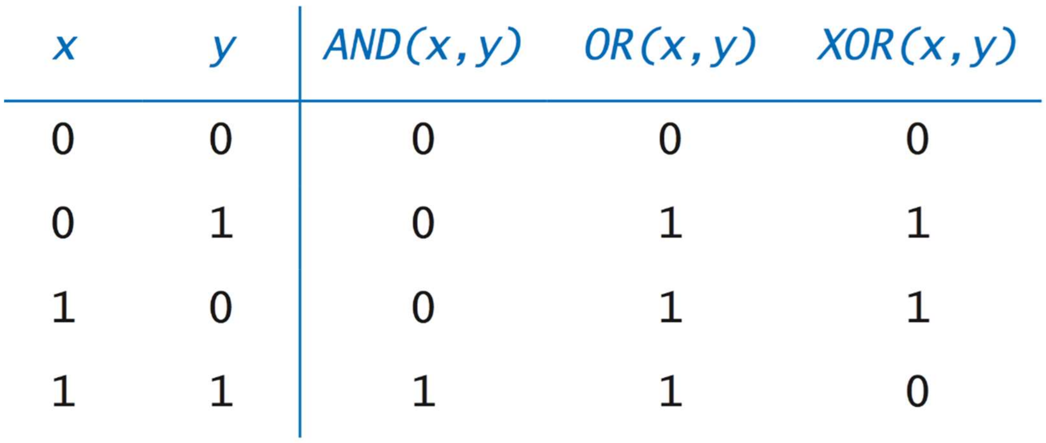
- & (and)
- << (shift left, multiply by powers of 2)
- >> (shift right, divide by powers of 2)
- ^ (xor)
- | (or)
- ~ (not)



Operator		Example	Explanation
<<	left shift	X = X<<2;	Before 0000 1111 X is 15 (8+4+2+1) After 0011 1100 X is 60 (32+16+8+4)
>>	right shift	X = X>>2;	Before 0000 1111 X is 15 (8+4+2+1) After 0000 0011 X is 3 (2+1)
&	bit-wise AND	X = X&28;	0000 1111 & 0001 1010 = 0000 1010 15 28 10
|	bit-wise OR	X = X | 28;	0000 1111 | 0001 1010 = 0001 1111 15 28 31
^	bit-wise XOR	X = X^28;	0000 1111 ^ 0001 1010 = 0001 0101 15 28 21
~	bit inversion	X = ~X;	Before 0000 1111 X is 15 (8+4+2+1) After 1111 0000 X is -2,147,483,633

Source: http://www.cs.uah.edu/~rcoleman/CS121/ClassTopics/Operators.html

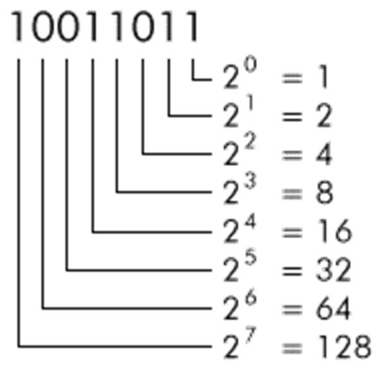
Working with an 8-bit processor, it will be necessary to learn your binary truth tables.



x	y	$AND(x,y)$	$OR(x,y)$	$XOR(x,y)$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Source: https://introcs.cs.princeton.edu/java/61data/

The link to Princeton goes into more detail about how numbers are stored in binary as well.



Source: http://www.pages.drexel.edu/~dj426/Lab_6.htm

In systems with more memory, you will still use all these bitwise operators, but you may be tempted to just use more variables rather than try to manipulate bits within a single variable.

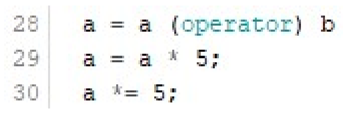
For example, your hero may have 8 different stats that can either be true or false. You could set up a variable for each of those stats. And that would take up 8 bytes of memory. No problem when your computer has 8 gigabytes of memory. But when your computer has 8 kilobytes of memory (or more specifically, 2 kilobytes as the Arduino has) you cannot waste so much memory. Instead of 8 bytes where 7 bits of each byte are unused (7 bits x 8 hero stats = 56 wasted bits / 8 = 7 wasted bytes of memory), you would use a single byte and check the bits of that byte to see the status of a particular stat.

The first practical application of this will be the controller which has 8 buttons. Rather than using 8 bytes, we use 1 byte and then reference 8 bits of that byte to set and get the buttons that are pressed.

Chapter 8: Compound Operators

- &= (compound bitwise and)
- *= (compound multiply)
- ++ (increment)
- += (compound add)
- -= (compound minus)
- -- (decrement)
- /= (compound multiply)
- ^= (compound bitwise xor)
- |= (compound bitwise or)

Compound operators are not just shorthand.



```
28    a = a (operator) b
29    a = a * 5;
30    a *= 5;
```

++ increases a variable by 1 and -- decreases a variable by 1

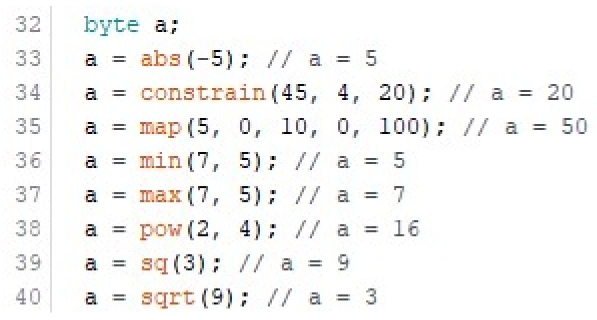
https://softwareengineering.stackexchange.com/questions/134118/why-are-shortcuts-like-x-y-considered-good-practice

Using the compound operators also helps the compiler create faster machine code. When using the long version, you are telling the compiler to move values around in memory, do a calculation and then move that value to the final memory location. With the compound operator, the operation is done directly on the value, avoiding having to move things around in memory. Line 30 will run faster than line 29.

When dealing with a low powered, 8-bit processor, every bit of optimization helps. When we start getting into game programming there will be a lot of math and although a single instance of this may not amount to much, in a game, these operations are run billions of times. So, a little slowdown quickly adds up.

Chapter 9: Math

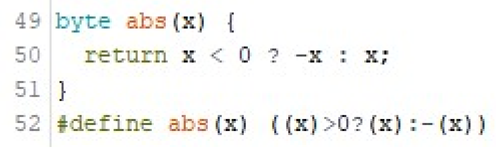
- abs()
- constrain()
- map()
- max()
- min()
- pow()
- sq()
- sqrt()
- cos()
- sin()
- tan()



```
byte a;
a = abs(-5); // a = 5
a = constrain(45, 4, 20); // a = 20
a = map(5, 0, 10, 0, 100); // a = 50
a = min(7, 5); // a = 5
a = max(7, 5); // a = 7
a = pow(2, 4); // a = 16
a = sq(3); // a = 9
a = sqrt(9); // a = 3
```

Absolute Value (abs)

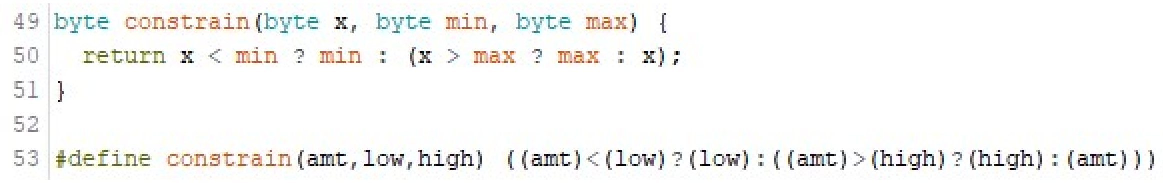
This function is set with a #define that simply checks to see if the number is greater than zero and return the number if it is, or the negative number if it isn't. If you attempt to create your own function with the same name you will be shown the #define as given in line 52.



```
49 byte abs(x) {
50   return x < 0 ? -x : x;
51 }
52 #define abs(x) ((x)>0?(x):-(x))
```

Constrain (constrain)

This function prevents the given value from going outside of the given range. If the number is less than the minimum, the minimum is returned. If the number is greater than the maximum, then the maximum is returned.

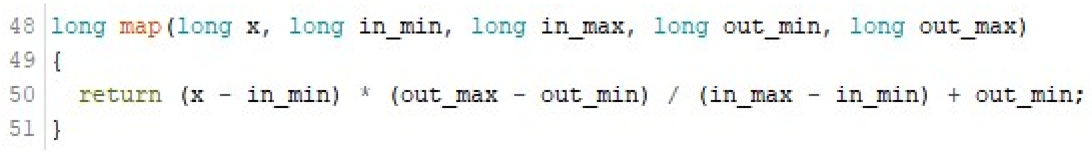


```
byte constrain(byte x, byte min, byte max) {
  return x < min ? min : (x > max ? max : x);
}

#define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
```

This function is defined with a #define. If you try to write your own function with the same name you get an error with the #define see in line 53.

Map (map)

The map function converts one range of values into another range of values.



```
48 long map(long x, long in_min, long in_max, long out_min, long out_max)
49 {
50   return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
51 }
```

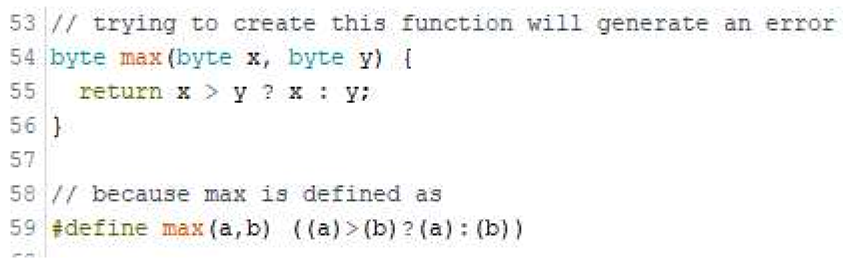
Source: https://www.arduino.cc/reference/en/language/functions/math/map/

Notice that the map function uses 32-bit integer math. This is problematic if you want to work with floating point numbers or want to stick to 8-bit math. A common use for the map function is texture mapping. This will tell you which pixel of the texture to use, given the relative position on the shape being textured.

Maximum (max)

Given two values, the function returns the larger of the two.

max is actually a macro.



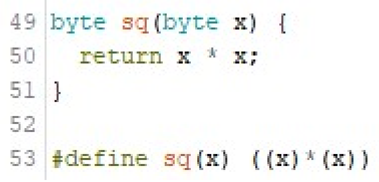
```
53 // trying to create this function will generate an error
54 byte max(byte x, byte y) {
55   return x > y ? x : y;
56 }
57
58 // because max is defined as
59 #define max(a,b) ((a)>(b)?(a):(b))
```

Minimum (min)

Given two values, the function returns the smaller of the two.

Min is defined the same way as max.

Square (sq)



```
49 byte sq(byte x) {
50   return x * x;
51 }
52
53 #define sq(x) ((x)*(x))
```

The sq function is another one which uses a #define to create it rather than defining an explicit function. If you try to create a function named sq, it will not compile and will give you the #define shown above.

Square root (sqrt)

This function returns the square root of the given number. It works using 32-bit math. It is very slow. Modern systems have a hardware square root function. The Arduino does not.

https://stackoverflow.com/questions/1100090/looking-for-an-efficient-integer-square-root-algorithm-for-arm-thumb2

There are several different approaches that can be used to find the square root faster but when dealing with a low-power 8-bit system, it is generally best to avoid the need.

The most common way to avoid using a square root which works in many situations is to simply compare squared values. If you need to see if object a is further away than object b, you can simply skip the square root since you are comparing relative distance, not actual distance.

COS, SIN, TAN

These are your standard Trigonometric functions which again, are not implemented in hardware and as such, are very slow. They use 32-bit math which further degrades performance.

What Did We Learn?

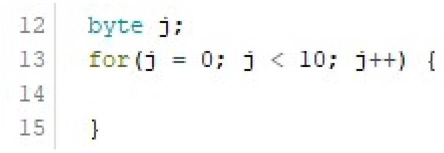
- The Arduino is not a number crunching machine
- Several math functions are created with a #define which avoids having to create specific functions for each primitive type.
- We will avoid floating point operations when creating games
- Some given functions need to be rewritten to be specific to 8-bit math if we want to use them

Chapter 10: Control Structures

- break
- continue
- do…while
- else
- for
- goto
- if
- return
- switch…case
- while

For

The for loop is used to repeat a section of code some fixed number of times. It has a starting condition, a loop condition and then an increment which can be negative or positive.



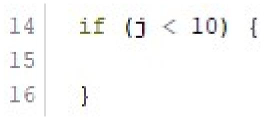
```
byte j;
for(j = 0; j < 10; j++) {

}
```

Whatever is inside the { } will be executed 10 times. The value of j will go from 0 to 9 and then, when j is 10, the condition will fail, and the loop will exit.

When using a for loop, it is not necessary that you start at any particular number, that the condition be any particular thing or that you increment any particular amount. You could also start at 10 and count down to zero.

If

An if statement takes a Boolean expression to determine whether the block of code inside of it should be executed. In other words, whatever is inside the condition must evaluate to true or false.



```
if (j < 10) {

}
```

Else

Else allows you to do one thing if the condition is true and another if the condition is false.

This

```
14  if (j < 10) {
15
16  }
17
18  if(j >= 10) {
19
20  }
```

Is the same as

```
22  if (j < 10) {
23
24  } else {
25
26  }
```

Break

A break command exits out of a control structure.

```
12  byte j;
13  for(j = 0; j < 10; j++) {
14    if(j == 4) {
15      break;
16    }
17  }
```

In this case, when j equals 4, the loop will break, and the program will continue at line 18. Often, a loop is used to search for something in an array and once it is found, the loop exits rather than continue through the rest of the array wasting time.

Continue

A continue returns the program to the top of the control structure.

```
12  byte j;
13  byte k = 0;
14  for(j = 0; j < 10; j++) {
15    if(j < 3) {
16      continue;
17    }
18    k++;
19
20    if(j == 4) {
21      break;
22    }
23  }
```

In this case, we will not reach line 18 until j is greater than or equal to 3. And once j is equal to 4, we go to line 24. So, we will only increment k twice.

At line 24, k will equal 2.

Using conditionals within a control structure allows us to work through something once and find multiple things.

Do…While

```
byte j = 0;
do {

  j++;
} while (j < 10);
```

You can see from the structure of the do / while control structure that it will execute whatever is in it at least once because the condition is not checked until after everything inside executes.

While

```
byte j = 0;
while (j < 10) {
  j++;
}
```

The only difference between a while and a do…while is the location of the condition check. A while loop may not execute at all if the condition is not met. A do…while loop always executes at least once.

Goto

It's generally better if you forget this control structure even exists. All the other control structures reduce to a JMP statement in assembly. GOTO is an alias of JMP. The problem with GOTO is that it makes the flow of your program difficult to read or understand and it makes your program difficult to modify. It is only used in low level error handling when writing device drivers. And even then, they are exceedingly rare as virtually any code that uses a goto can be rewritten to not use it and result in much easier to read code.

```
label_1:
  j++;
  if(j > 10) {
    goto label_2;
  }
  goto label_1;

label_2:
  Serial.println(j);
```

Note: you cannot put a label just before a closing bracket. A label must be put before some other statement. A goto has to go somewhere.

Return

The return keyword is used to return from a function. It can be used to return a value out of a function.

```
byte add(byte a, byte b) {
  return a + b;
}
```

Notice that the type of value that is being returned must be specified before the function name.

If you function does not return a value, then you would use void.

```
void foo() {
  return;
}
```

Notice that you can still use a return, but it cannot be followed by any variable or value. You cannot return a value of a different type than the function is defined as.

Switch…Case

A switch statement is used to check a variable against a list of values.

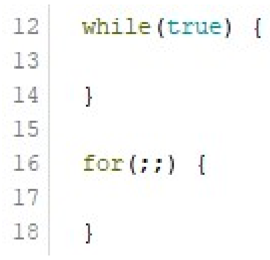
```
  if (j == 1) {

  } else {
    if (j == 2) {

    } else {
      if (j == 3) {

      }
    }
  }
```

Can be replaced with

```
  switch (j) {
    case 1:
      break;
    case 2:
      break;
    case 3:
      break;
  }
```

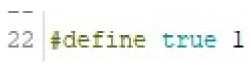
Infinite Loops

https://stackoverflow.com/questions/2611246/is-for-faster-than-while-true-if-not-why-do-people-use-it



```
while(true) {

}

for(;;) {

}
```

Both control functions are identical. However, some prefer using for because it is something that is consistent across all languages. The while version of an infinite loop can use any non-zero value. “True” is just a macro (#define) for 1.



```
#define true 1
```

You may recall from Chapter 1 that on the Arduino, the loop function is called within an infinite loop in the main function.

All loops require an exit condition.

On the Arduino, the exit condition for the main infinite loop is the power switch.