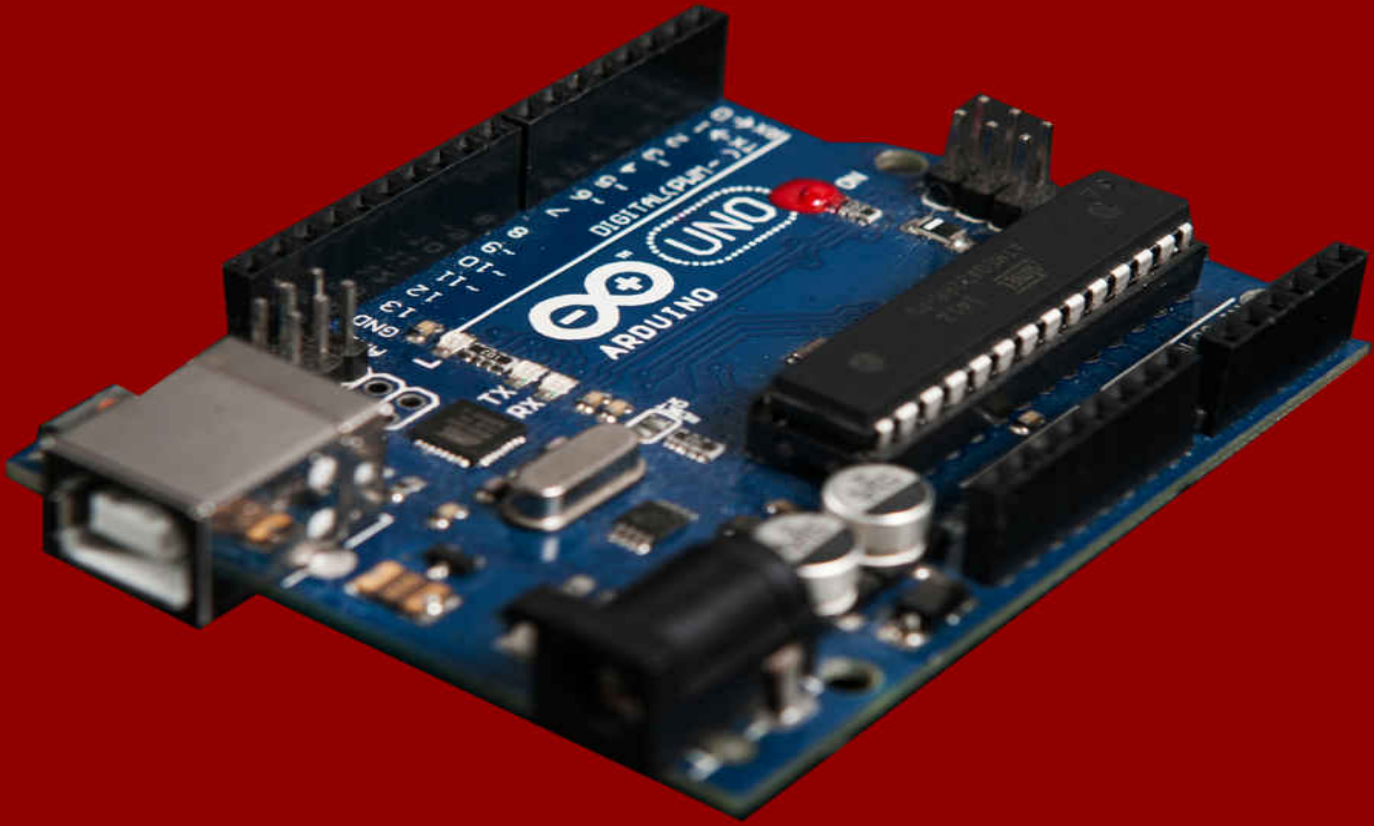
ARDUINO

BEST PRACTICES TO EXCEL WHILE LEARNING ARDUINO PROGRAMMING



MILES PRICE

Arduino

Best Practices to Excel While Learning Arduino Programming

Miles Price

Table of Contents

Introduction

Congratulations on downloading *Arduino: Best Practices to Excel While Learning Arduino Programming* and thank you for doing so.

Learning programming can be quite difficult. The purpose of this book is to act as a companion book to other books which have a focus on actually teaching you about Arduino programming. This book will focus in some capacity on larger programming concepts to make sure that you understand them, as well as how these concepts interconnect with Arduino and how you can take advantage of them to become a better programmer.

It will also give solid general advice for you as you attempt to become an Arduino programmer, such as things to help bolster your learning experience altogether. Assistance will be given in finding communities to help you out, and a long list of different tutorials and sample projects for you to either try out for yourself, emulate, or simply to act as inspiration for your own ambitions.

The truth is that learning programming can be challenging and daunting, and there is generally a lot to take in. Because of this, it’s important that you have some sort of reference material that will help you to become a better programmer in the end and that will help you when things start to get a little heavy. It can be very easy to start to drown in the details when you’re starting to program, especially with something as nuanced as Arduino.

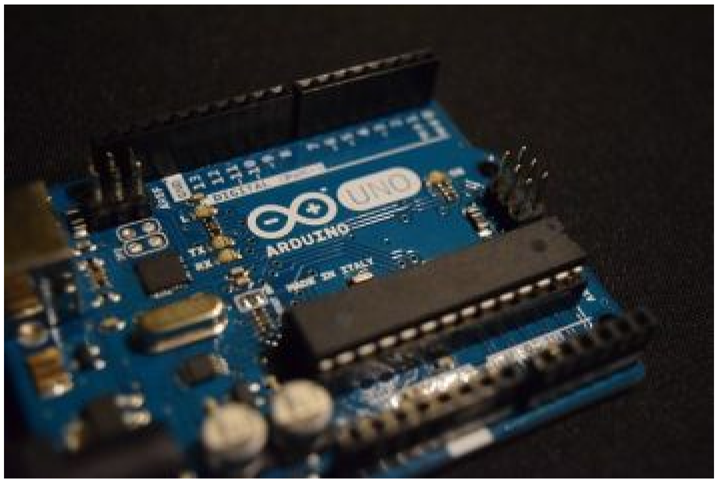
Nevertheless, with this book in your hand, you’re going to develop a firm grasp on a lot of different concepts and start to feel like you have a very solid idea of everything that you need to know to keep moving forward as a programmer. Along the way, you’ll also build up a large base of support in the form of other programmers that are doing exactly what you’re doing: trying to get better.

So relax - Arduino can be frustrating, and so can programming. However, with this book in your hand, you’re going to have plenty of educational material to help you figure out exactly what you’re doing as you go along.

There are plenty of books on this subject on the market, thanks again for choosing this one! Every effort was made to ensure it is full of as much useful information as possible, please enjoy!

Chapter 1: Learn About Arduino

This is perhaps the best starting place for any. The best way to get started with learning about Arduino and learning how to be a capable Arduino programmer is to learn, beyond face value, what Arduino actually is, as well as what purpose it serves and what purpose it's necessarily *supposed* to serve. If you want to be a better Arduino programmer, then having some sort of idea of the origins of the thing itself will open your imagination up to the things that you can do with it.



More or less, Arduino first came around in the early 2000s when students at a design institute in Italy were trying to develop a low-cost microcomputer that would allow people to pursue many technical projects that would normally be prohibitively difficult or expensive for people who are less familiar with advanced electronics.

This was coupled with the development of easy to understand platforms that would allow people who otherwise would never really touch such complex hardware to have the opportunity to learn more about the complexity of hardware and software in an intuitive manner that would have a relatively low learning curve and enable them to get into tinkering as a hobby.

What resulted was a tool that serves as an amazing prototyping device. The Arduino is capable of allowing people to build easy prototypes of electronic inventions at little cost in terms of components. It also is relatively forgiving of a platform and allows people the room to experiment and tinker in a relatively low-stakes environment. At the end of the day, too, if something does go awry, the Arduino is low-cost enough that it’s easy to replace.

However, the Arduino isn’t just good for prototyping. Hobbyists would quickly pick it up and see it as a great catalyst for all sorts of projects that otherwise would be difficult to do. For example, there are projects such as a colored keypad that unlocks doors, which seems at first glance like something straight out of a sci-fi movie.

Basically what Arduino has allowed people to do is to take any gadget idea they may have and tinker with code and hardware until they have something that works for them in terms of their overall vision. In this capacity, there are few things better than Arduino.

Moreover, the actual language for working with the Arduino is relatively simple; many of the electronics which are compatible with the Arduino are very easy to use, and there is a large enough community that you can find support for almost any issue that you may have. (We’ll talk more about this a bit later, though!)

All in all, in terms of a rapid prototyping tool, it is very difficult to beat the Arduino. However, it’s also difficult to beat it in terms of a cost-effective solution for any gadget or trinket ideas that you may have floating around in your head.

Chapter 2: Join Arduino Communities

There's little better that you can do for yourself as a new programmer than to join communities; there are few things for which this is more applicable than Arduino.

The simple truth is that Arduino is not easy in any way, shape, or form. In the hands of a hobbyist who is willing to learn, it can be an extremely powerful and interesting contraption capable of making your nerdiest dreams come true. However, this does have a precondition - you must be willing to *learn*.

However, not all learning comes from a book; in fact, most doesn't. You could spend all day poring through the pages of this or that book about Arduino or Arduino projects and still not really pick up much information if you aren't actively trying them out.

More than that, there's not really a way to ask a book a question when you get stumped. And believe me right now, you will get stumped: when you're working with Arduino, you're working with electrical circuits and a bunch of small doo-dads that you've most likely never tinkered with before. These are subjects that people quite literally go to college for and spend years studying, so it's not exactly to be expected that somebody who just picks up a board on the internet will be able to use it perfectly right away. On top of this fact, you also have to learn to program in one language or another if you want everything to go smoothly and you don’t just want to rely on other people’s code and their exact build paths and instructions.

Because of these reasons, there’s no greater decision that you can make than to spend some time researching Arduino communities and joining one or two. There are several perks that come from joining Arduino communities.

The first is a constant source of inspiration. People are working on Arduino projects all the time, and there are constantly projects released that build up on the idea of an older project. This means that there’s a natural sort of linear progression to the world of Arduino, and the people within it are deeply embedded in the hacking subculture. The hacking subculture is largely based on freedom of information and sharing what you make with everybody such that anybody else who enjoys tinkering can likewise start tinkering using something you made as one of their starting points. This works out for you in two ways.

It works out for you on one hand because you don’t have to worry about seeking out inspiration. There are a huge number of projects that are there to show you what can be done and to help you imagine what *could* be done.

It works out for you in another way in that if any of the projects do particularly inspire you - which they will - you generally can find a lot of information by the people who created the projects which will divine to you the specific path that they took. When you’re just starting out with Arduino programming and conceptualizing your Arduino projects in general, this can be a pretty major deal, and it can be a major source of help.

The second is a constant source of feedback and help. As I said earlier in the chapter, there is no way to ask a book for help if you get confused, and if it doesn’t adequately explain to you a certain topic, then you most likely aren’t going to be receiving any additional information on that topic from the book. By having a firm idea of what communities are available for you to browse and also being a member of those communities, you can first search when you have a question and then ask if you don’t manage to find a satisfactory answer. As I said before, freedom of information and helping newcomers out is a pretty major part of the hacking subculture, so people generally are very welcoming of questions from interested individuals.

Moreover, there aren’t many better learning methods than learning from mistakes. If you start to integrate into some communities based on Arduino and programming, then people there will be happy to let you know when you could be doing something better, generally in a relatively tactful way. It is through these mistakes you make that you’ll grow as a programmer and develop a far better technique than you would have had otherwise.

However, it is not only about learning from your own mistakes but also from the mistakes of others. For example, if you were to take the time to read other people’s posts and look into their projects to see what they were doing and what others had to say about it, especially posts when somebody can’t get something to work, you’re going to learn a lot by analyzing their code as well as people’s responses to their code. You’ll be subconsciously learning the best methods for programming with very little effort on your part, just by simply paying attention to what others are doing and how others say that they could be doing it better.

The best part of all of this is that Arduino has a pretty massive following. A lot of people work with this technology, and every day, there are more who develop an interest in working with it. Because of this, there are a massive number of different forums out there dedicated to helping people like yourself either become inspired and post their projects or get help from others and in general, build a sense of community.

One of the most common is Hackster. Hackster is a place where people post their Arduino inventions, and people can react to them. Hackaday is a very common one as well. Either of these will give you a huge number of ideas and a generally large amount of experience looking at and understanding Arduino projects. As I said before, the projects generally also will be bundled with code and precise directions, should you want to either do one yourself or modify it for your own devices.

In terms of forums, there are a lot of options. One of the most popular and active is the forum hosted by Arduino themselves on the Arduino website. Here, people congregate and discuss everything from questions about programming to general guidance and advice on projects all the way over to general questions about electronics. There are even off-topic discussion sections where users can build camaraderie and rapport with one another outside of the contexts of pure tech babble (as fun as pure tech babble can be.) This means that the forum can in a way become your home away from home when you’re working on Arduino projects, a keen source of guidance and friendship both as you attempt to pick up a daunting new hobby.

Chapter 3: Learn the True Logic of Programming and Hardware

In the previous chapter, we talked a lot about what you can do to build yourself a support net, as well as the importance of *doing* so when you're trying to learn programming, especially with something so fickle as Arduino. In this chapter, we're going to focus on really reinforcing your knowledge of programming and hardware, as well as how they intertwine in terms of Arduino. This will enable you to make better programs because you'll have a much greater understanding of just what's going on under the hood in them.

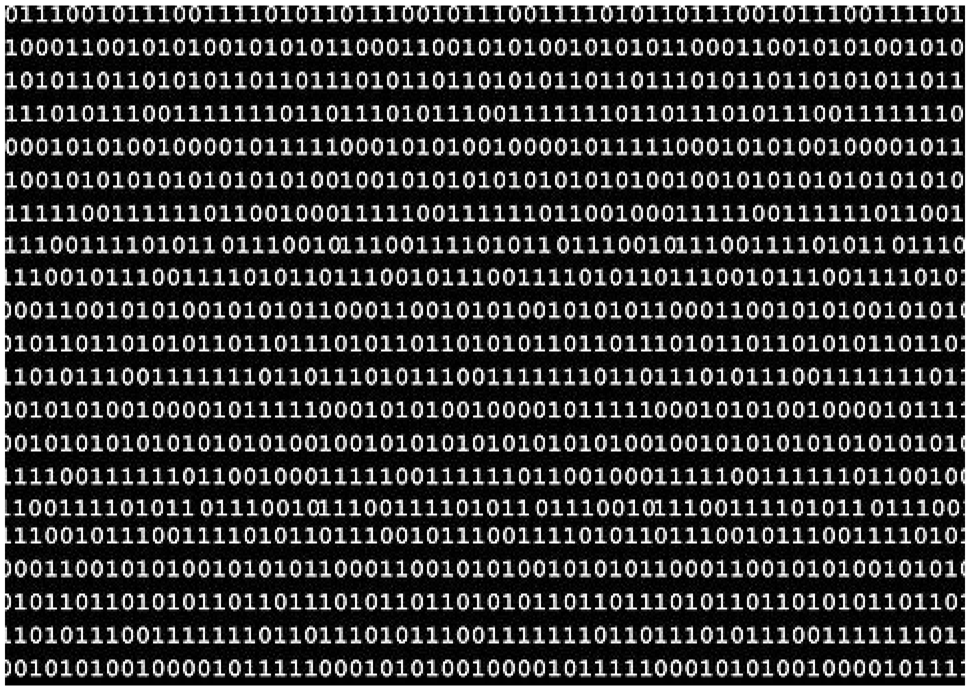
Since you're working with Arduino, the chances are good that you're most likely working with C and C++, since these are the languages that the core Arduino library builds upon. However, since there are APIs intended to make pretty much any programming language that you can think of Arduino compatible, the lessons taught in this chapter will be purposefully vague so as to be applicable to any given language that you may be working with. Again, the purpose of this chapter is a reinforcement of main ideas and giving you new ways to think about things that you potentially hadn't before. In other words, where you may know vague programming essentials, the point of this chapter is to bolster your knowledge and teach you even more than you already know. This is all an element of building up a stronger framework within which you can work.

How Computers Work With Data

The first thing that we're going to talk about is how computers work with data. In this section, we're also going to be discussing the concept of memory and how it correlates to computers. It's an important topic worth diving into, especially when you're working with low-level systems.

Before you do anything else with a computer, there's one very, very important idea that you need to understand - everything that a computer does is one form of data manipulation or another. Even something so simple and plainly causal as the typing of a character into a word processor and the display of that character on your screen within the word processor. What you don't realize at any given point while doing so is that the program is doing a lot under the hood which you aren't recognizing; for example, when a character is entered into a word processor, the word processor's active *state* gets changed.

All programs work within a given *state*. This state reflects how the program is at any given moment, and any program's state is the result of various different things happening at the level of memory and processing. The most obvious is that the various states are stored as data within the computer.



There is not one application in the world that is not built upon changing states; such is simply impossible. The program would be practically useless, and even the launching of the program would be a change of state within the operating system, even if not within the program itself. Moreover, variables that affect the state outside the program are altered constantly, so the state of the program at 10:49:36 PM is fundamentally separate in terms of instance from the state at 10:49:37 PM by simple cause of the fact that this is an altered variable at some point on the computer. It is impossible, except at the most rudimentary of levels - the turning off and on of an LED light by a button press through an Arduino program, for example - to escape the reality of changes of state. Even Arduino constantly reflects changes of program state through the fact that it has a constantly looping function within it.

So how do computers view states? What is a state, and how small can it go? In a rather abstract sense, states can go as low as the singular variable - for example, any variable has a given state that it remains as until altered.

What maintains these states? What determines these states? A greater understanding of programming at any level requires a thorough answer to these two questions.

The first thing that we need to talk about is the maintenance of data and states. How are these maintained? The simple answer is that these are maintained through storage, saved to be later manipulated by the program. There are two main methods of state and value maintenance, where a value is inherently considered a state because it’s a static representation of some sort of data: these are the *hard drive*, which is used for long-term storage of data that will need to be accessed at a later point, and *random access memory*, which is used for the storage of short-term variables or variables which don’t necessarily need to be stored on the hard drive.

Understanding how random access memory works is a major part of learning to be a better programmer, and doing so can, in fact, be a major boon to your ability to program ably. Random access memory essentially works by setting apart extremely small sections of data that can be altered in size and allocated as need be, as well as freed up and dismissed. It is the prime catalyst to actually being able to work with and manipulate things in your programs.

Perhaps a better question is how this applies to Arduino programming. However, this really doesn't need much of an explanation; when you're working with electronics, you are generally working with states. For example, an LED light has two states - on and off.

Moreover, an in-depth program - for example, one which is designed to read vital levels from a plant and then triggers the release of water accordingly - will be massively utilizing data, which is then intended to be manipulated either by the programmer or by the program itself.

So how does one build on this sort of notion? And more than that, what makes it important to us as potential Arduino programmers?

One of the primary ways in which this is important is the fact that, again, you are most likely using C and C++. The very nature of these languages is very low-level, meaning you have the ability to work directly with memory, to send pointers around, and so forth.

If, for the record, you don't know what pointers are - pointers are essentially things which *point* to a direct place in the computer's memory so that the data which is stored there may be altered directly. It's a relatively simple concept, but it can be hard to grasp at first.

However, beyond that, when you're working with the Arduino, you're not working with a very sophisticated system; knowing how data is manipulated within the system and, moreover, how it's stored within the confines of the system can lead to you have a better understanding of the manifold ways in which you can actually treat and manipulate the data therein.

Values and Variables

At this point, we're going to drive home a few factors about a couple key programming concepts. The first of these is *values* and *variables*. Often, programming books will teach you data types but not give you a lot of information as to what any of that actually means.

Given that the most prominent languages available for use today, especially within the context of Arduino programming, are either directly C, and C++ or C/C++ derived (such as Python, Java, C#, and so forth), we're going to be looking within the context of C-style values and C-style variables, as well as how both of these are handled.

In the last section, we looked into a bit more detail as to how computers handle data. What we didn't discuss is *why* they handle it in such a way. There is a relatively simple answer to this, though: computers cannot think. They cannot parse, and they cannot reason.

Computers are good at what they are named after: computing. And, in effect, this is essentially all that they are really doing at any given point: handling various computations and moving this or that value around to make some sort of change in the context of the larger program.

In the end, though, things get a bit more complicated than that: computers don't really have any concept of anything other *than* numbers unless you create an interface that allows them to have a concept of things at a level greater than this. This interface has been created many times over, by now, but that still doesn't necessarily mean that at the actual deep-down level of the computer that it understands it in any greater capacity than this.

However, computers don't even really understand things in terms of numbers: they understand them in terms of binary code. That is, a sequence of ones and zeroes that are understood by the computer to represent certain numerical values. The computers can parse these binary codes and perform mathematical operations on these binary codes. This creates the crux of the computer's processing, which is carried out by what is known as the central processing unit or *CPU*.

Programming languages, even at their most basic level (which is assembly programming, the language used to speak directly to the computer's processor and RAM), are designed to make it so that people don't have to simply work within the confines of binary code. More and more intricate systems are developed, and these systems define new systems which then reflexively define new systems still.

This is true especially for languages such as C and C++, as well as their derivatives in turn. These languages actually are renowned for their closeness to the computer's internals despite the fact that they are relatively easy for people to read. This can make it easy for one to work with them and really creates an abstract level for the programmer to work with.

However, this doesn't mean that these underlying concepts still aren't being practiced, they're simply being masked. So when you work with data and values, the computer is still parsing the information that you're sending it as more pertinent and complicated information than it's leading you to believe. All things will be eventually boiled down to ones and zeroes.

All things, too, within a computer, correlate to three things important: a number, a binary equivalent, and a space in the computer's memory.

This is where this aspect of the lesson starts to build on the previous part of this chapter. An intricate understanding of the interplay of all of these concepts will allow you to really start to comprehend the multiple levels at play in terms of a computer's internals, which will then transfer to your Arduino programming with relative ease.

So let's talk about this a bit more: when I say that all computers work within these confines, what I mean is that all values that your computer might interpret are eventually understood in this sort of manner as well. That is to say that if you were to use a *char* variable, your computer would interpret the ASCII character not ultimately as a character, but first as a character, then as assembly code, then as a number, then as binary.

These binary things are stored in terms of things called bits. Bits are the storage medium of your computer and are the way in which data is held and managed. Your random access memory automatically stores and recognizes things in terms of bits.

So what is the key point of all of this? That your computer doesn't actually understand what you're working with. This is why one data type is often incompatible with another data type, for example, or why certain data types can hold more data. Take, for example, the *integer* against the *long*.

The integer, in C programming, is 4 bits per the standard. This means that the binary code which can be stored in that bit can represent numbers up to and as large as around 32,000, with the capability for negative numbers being the same numbers being conversely negative. Meanwhile, the *long*, though still an integer, has more bits of storage reserved to it - 16 bits, exactly. This means that the binary configuration that can be stored for this number is larger, meaning that larger values can ultimately be used and determined here.

Some other things are taken into account, as well - unsigned values eschew one spectrum of binary numbers for another, despite taking up the same amount of storage in terms of bit space.

Understanding this is a key milestone in understanding how and why computers parse data in the way that they do.

This raises another question - what is a *variable*? The book will make the assumption that the reader knows what a variable *is*, but do you know how a variable truly *functions*?

A variable is, more or less, a reservation of space. When you declare a variable, you set aside a bit space of a certain designated size within RAM, the size being designated by the type of the variable. This is why you can often cast something of a smaller bit size data type to a larger bit size data type without corrupting the information, but you generally cannot cast something of a larger bit size data type to a smaller bit size data type without having the value change. (A cast, if you are unaware, is the changing of a variable's data type to another.)

When you refer to the variable, you refer to that specific location in the computer's memory, or rather the value which is stored at that location.

So what does all of this mean to you as a programmer, and what’s the point of going in-depth on this? Well, the first is that there is the off-chance that you aren’t coming from a language where data types are actually explained. For example, in a language like C++ or C, you are required to state data types, and they are, in fact, a major part of programming in these languages. If you don’t understand how data types work in a conceptual way and their relation to memory, then you aren’t really going to understand a lot of what’s going on with your variables in these languages.

More than that, a lot of books - as I said - will gloss over this sort of important information in favor of a much lighter covering on the topic which doesn’t actually tell you a whole lot in the way of understanding the overarching topic of data types and variables. Even still, there are some things that aren’t discussed here because I’m trying to stay language-neutral and just give a theoretical understanding of these topics. One such example would be the way that variables are passed between functions in C/C++ versus other languages, where variable values are copied, but the address of the variable itself isn’t actually passed in C/C++. This means that an altered variable in another method in C/C++ will lead to the true value of the variable not actually changing, where in a language like Python, for example, the variables will be passed in reference to their address, meaning that a changed variable in another function changes the variable in the original function.

Truth and Logic

Here we get into lessons which are actually rather foundational to programming but that aren’t very much discussed: logic and truth. If you want to be a professional programmer and get a degree, most degree plans will require you to study discrete math for the reason that discrete mathematics discusses many different topics which are core to computer science in a rather abstract way, like trees and graphs and, above all, logic.

All things which are carried out by a computer are ultimately just methods undertaken to verify whether something is true or whether something is false. After all, this is all an equation is in the end - the discovery of truth on one side of an equation in terms of the question on the other side, which makes math almost linguistic in its properties.

Having a keen understanding of logic and truth is imperative if you want to work on large projects that go deeper than simply pushing a button and having something happen. It’s true too that when you’re working with truth and logic that you start to see patterns that you might not otherwise.

In the end, having a solid understanding of truth and logic is what really causes you to *think like a programmer*. Programmers are constantly thinking of how something can be done or interpreted in a logical and concise manner, rather than getting bogged down with details of aesthetics. Programmers are, in that sense, pragmatists, such that they want their programs to do exactly what they need to do and they don’t want them to waste their time.

This is hardwired into the programming tradition by virtue of the fact that computers solely understand logic and truth. At the end of the day, they are mathematical machines, and math is comprised of nothing but these two concepts. The ability to interpret, expand, and work within these two concepts is absolutely paramount to becoming a good programmer.

The first thing that we should talk about is logical statements and what they are, as well as how they work.

All statements have some degree of logic to them, and all statements can be parsed in a logical way provided that they are semantically sound. The idea of logic studies is to reduce the semantics of a sentence or phrase to those which are most necessary to convey a key point, then analyze it to say if it is effective at doing so.

Take, for example, this sentence:

If I pay you $20, you will wash my car.

This sentence is very semantically sound. It is composed of the premise and the conclusion - *if I pay you $20* is the premise which implies the conclusion *you will wash my car*.

Let’s look at yet another example:

If it rains today, I will not go to the park.

This sentence too is semantically sound and rather barebones. There is an implied *then* after the comma in both of these example sentences. Either sentence is based on the idea of a premise which implies a conclusion.

This is the basis of logical statements in programming, but not necessarily logical operations.

Note that with both of these, we can take the premise and then break it down even further in a logical sense so that we can actually analyze the true meaning of the sentences. Let’s look at the first.

If I pay you $20, you will wash my car.

First, let’s detach the premise to analyze it:

If I pay you $20…

Now, let’s take the *if* clause out of it:

I pay you $20.

This is the core of the statement. There are a couple implications here, first that there is a *set amount to be paid* which is equivalent to $20, the second that the amount *is being paid*.

Understand that this relation of subject to action is a key part of logical statements and logical premises. Using these, we can set up nearly Socratic logical statements:

If I pay you $20, you will wash my car.
I paid you $20.
Therefore, you wash my car.

Note that this only sets up a unilateral relationship between the premise and the conclusion. It means that if the premise occurs, the conclusion *will* happen. However, in logic and computers alike, it doesn’t necessarily mean that the conclusion won’t just happen anyway. For example, somebody may just start cleaning my car without paying them; because there is no established relationship there, they are free to do so. However, there *is* an established relationship anchoring the premise to the conclusion, even though there isn’t one going the other way. In this capacity, if I pay somebody $20, they will wash my car per our agreement. They may wash my car even if I don’t pay them $20, but if I pay them $20, they may not *not* wash my car.

The premise here can be looked at in both programming and mathematical terms as an *expression*. An expression is a statement which necessarily implies some sort of relationship between two things.

From a programming standpoint, we can look at this in one of two ways. The first is that we take the statement in a literalistic way:

```
if (amountPaid == 20)
        you.washCar();
```

However, this isn't the correct way to break it down logically. This only sets up a relationship according to the *amount paid*, not necessarily to the action taking place at all. We can assume that this isn't a time for a numerical expression but rather for a boolean expression:

```
if (I.havePaid == true)
        you.washCar();
```

So this illustrates an important lesson: logic is not easy, and sometimes the most immediate solution when analyzing a premise isn't the necessary way forward.

Let's look at a statement where we *can* distill a premise into a numerically comparative expression. Let's take the statement *if I have zero cans of Coke, I will go to the store.*

First, note once more the way in which this sentence functions logically: the premise sets an anchor to the conclusion, but the conclusion may occur with or without the premise. I have only set up a situation where if I'm out of Coke, I must go to the store; however, I am not *required* to have run out of Coke to go to the store, and I may still go to the store anyway.

Let's parse the premise again. There seems to be only one clear way forward:

```
if (fridge.cansOfCoke == 0)
        I.goToStore();
```

This is a case where numeric analysis of a given premise is the right way forward. This also leaves the statement open-ended.

This is where we have to discuss another topic that’s often underlooked: the usage of if statements in terms of logic and program flow. If statements are an incredible utility, and this book makes the assumption that you know in a proper sense how they work. It also makes the assumption that you know how *else* statements work. I’ve always referred to the difference between the two as a *passive conditional* versus an *active conditional*.

The difference between these two is of great importance when it comes to logical flow, and it seems like nobody ever takes a methodical approach to discussing when and where to use one or the other. This can be of great confusion to new programmers who don’t quite understand the structure of logic yet - especially when younger - and don’t seem to have their heads on, so to speak, when it comes to questions of optimal logical programmatic flow.

Passive conditionals are so-called because in no way does the logical flow of the program depend upon them. In one way or another, they are completely and entirely exempted from the normal logical flow of the program. That is to say that if a certain condition isn’t met, a passive conditional - in reference to a conditional which only consists of a given if statement - may be completely skipped over by the program. The logical flow of the program would then bear no consequence as to whether the given statement even existed in the first place.

These have their uses, largely in terms of catching single conditions that should initiate a given action. Later, in the algorithmic section, you’ll see one such event: a variable should be swapped with another, but only in the case that the first variable is larger. If this isn’t the case, however, we don’t actually want our program to do anything.

These so-called event triggers are important to flow in many programs. In video games, for instance, keystrokes are listened for and reacted to with event triggers the vast majority of the time. Other chain reactions within the program are handled methodically through if/else statements and entity states to handle things like animation in respect to these keystrokes.

In Arduino programs, passive conditionals find a lot of use in the sense that states are used rather extensively. Take the state of a given button connected to a switch. Were this button pressed and in the on position, a current would be given. Arduino would, therefore, register the button as being in the *on* state. One could implement in Arduino's running loop that if this button is in the on position, something may happen.

Active conditionals, on the other hand, are intrinsically attached to the logical flow of the program. Active conditionals are if else statements or if else if else statements; in other words, statements which have some sort of backup clause should the given conditions be found to be false.

These are great for making logical decisions with your programs. They allow you to start decision branches and are a key foundation to things such as neural networks. Decision branches give your program a crazily high amount of interactivity in a logical sense and make it able to do way more than it would be able to otherwise.

Understanding when to use a passive or an active conditional is paramount to being a great programmer, but not every book takes care to make sure that you understand the logical differences between the two; they often will rush through this important part to get to discuss other things.

Loops

Loops are yet another thing which are incredibly important and foundational but are often understated in ordinary programming books. The thing is that the authors of programming books tend to, in my opinion, undervalue the weight that these concepts have in true relation to the gravity of the program. It's for this reason that I'm going to spend as much time as I can expanding upon these concepts so that you understand when these things are to be used in terms of your program's flow and direction.

So, let’s start by talking about loops in a rather abstract way. Loops are very much a common occurrence, but we don’t always appreciate their utility. Loops are prevalent in things which happen every single day and, indeed, in most actions that you take. Any sort of repetitive action - repetitive in the sense of literally repeating, even if you don’t notice it - is a form of a loop.

Let’s consider something as innocuous as scrolling down your Facebook feed. Even this is a loop in a certain way. You look at a post, then you evaluate the post, determine whether you want to like it or comment on it (or block the person who made it), then you move on to the next post. The process repeats ad infinitum until for one reason or another, you decide that you’d no longer like to look at Facebook posts. The loop is therefore considered complete.

Loop logic applies to a whole lot of different things that we never quite take the time to appreciate. However, appreciation of loop logic in its most raw form is important if you want to become a better programmer. Again, the purpose of this chapter is to look at things which are normally glossed over by other programming books.

Before we think on the logic of every loop, though, let’s just delineate what loops there *are*. There are *while*, *do while*, and *for loops*. In languages such as C#, there are also *for each* loops. All of these loops serve diametrically different purposes within the context of program logic, but from a detached view, they tend to look and act very similarly. It’s only through experience that one really realizes the difference between these loops in a logical sense.

However, I'm going to try to explain the difference between these loops in a logical sense because any large-scale project that you do is inevitably going to involve loop logic and it's important that you as a programmer can comprehend which you should be using and when. Additionally, if you are a new programmer than you may not even realize that these loops can be used in such a way as they're about to be presented, or you may not have even thought of them that much. In this sense, this portion will give you the information necessary to think like a more seasoned programmer.

Let's start with while loops. A lot of introductory programming courses like to say that while loops and for loops are interchangeable because they aren't willing to bog you down with information about the practical usage of the two; while they are, in many cases (especially when you're starting out) interchangeable, there are many cases where they are *not*, in fact, interchangeable, or at the very least shouldn't be interchangeable in the name of good practice.

This is because the real-life application of these loops is diametrically different from the textbook application of these loops.

Let's take while loops, for instance. While loops, in terms of logical function, are simple. They have a chunk of code within them which will execute repeatedly until the condition given is no longer true. Hence, the code runs *while* the condition is true.

```
while (condition) {
        // code within
}
```

This is pretty simple at face value, but it can be a bit denser of a topic when you consider its actual utility. For most functions where you're iterating, a for function would carry far more utility. So what good does the while loop actually do?

Well, the utility of the while loop comes through in the fact that it can be used efficiently and easily to constantly check a *boolean* condition. That is, it shouldn't be used for iteration but rather for carrying out a specific action for as long as something else *hasn't* happened.

This kind of logic is often called the *game loop*. This is a reference to the fact that in games, a similar procedure will often take place over and over until a certain win or loss condition is met, at which point the game loop is considered over (because the end condition has been met.)

Meanwhile, while functionally similar, the *while* loop actually differs a lot in practical usage from the *do while* loop. While the do while loop is frowned upon in some circles as being a poor choice in terms of programming convention, it still serves a functional purpose. The do while loop is used to execute a certain block of code, then continue executing it for as long as the given condition is met.

The thing is that much of the time, the condition *won't* be met in practical usage, so this actually differs quite a bit from the while loop. More often, the do while loop is used generally when the condition needs to act as a safeguard. The code *must* execute once, but if something goes awry or isn't quite as expected, the code will continue to execute until whatever issue is ironed it.

```
do {
        // code goes inside
} while (condition);
```

And both of these differ largely in practical usage from the for loop. The for loop is used to iterate over sets, especially arrays, whether those be the arrays themselves (e.g., an array of integers) or an array within a class definition (e.g., iterating through single characters in a string or iterating through a string array after a string split function). The for loop sees a great amount of its utility in the fact that it is specifically intended for iteration.

Languages such as Python make this perfectly clear in the fact that their for loop structure actually demands a range to iterate through. There are a number of other languages which follow this example. Unfortunately, most C-style languages typically don't make the cut, leaving themselves a tad bit vague.

However, if you really break down the declaration of a for loop in a C-style language, you begin to see its utility as an iterative loop. First, you actually declare an iterative variable. This holds a massive parallel to the declaration of an iterative variable in Python for loops, but it’s much clearer in Python what the exact intention of the iterative variable is. Afterward, you define the condition for which the loop runs. This is where you get to define your effective range as the difference between the maximum end point of your loop and the minimum start point denoted by your iterative variable. You then can designate an iterative step, which is normally ++ or --, though could really be anything so long as it’s something which moves at the same rate every time.

This gives us the following structure for for loops:

```
for (iterative variable declaration; condition; iterative step) {
        // code within
}
```

While Pythonic languages have the somewhat clearer variant like so:

```
for iterator in range_definition:
        # code goes here
```

For loops generally serve a common purpose across languages, though, which is the key point to take home; most languages have some form of a for loop. The thing that differs is the implementation of the range definition.

For loops, too, have their limits; this is where *for each* loops come into play. For each loops are used to iterate through high-level objects. They’re often used in conjunction with high-level collections such as lists, vectors, or dictionaries. They are composed of an iterator object of a given type; as the collection is iterated through, you may then use the iterator object as a placeholder to access member data of every element within the given collection. This is wildly useful but often understated. It also bears a strict logical difference from the other forms of loops, so it is never given its fair time. The general syntax of a foreach loop is like so:

```
for/foreach Object o in List {
	// code goes here
}
```

While some languages manage to get by without a foreach loop, they do massively simplify the process of iterating through objects, as well as ensuring that things go smoothly within the loop in terms of object definitions.

Object-Oriented Programming

This book assumes that you know things about programming, or at the very least have another source to learn them from and are using it as a supplement. Because of this, while it would be very simple and perhaps even easy to start rambling on and on about what object-oriented programming is and what objects and classes are and so forth, this book is written based on the assumption that you know what they are.

Rather, the intent is to help you to become a better programmer in general by explaining the logical and theoretical side to things that are often skipped over. This pattern isn't going to change for this section.

Object-oriented programming is, nonetheless, quite important, and it's important that you have some sort of understanding as to what its purpose is in terms of programming paradigms, as well as the general design philosophy that goes into it.

You may be wondering, if you have any familiarity with object-oriented programing, why one would be interested in it as an Arduino programmer; after all, Arduino is procedural by its very nature… right?

Wrong, actually. More nuanced and sophisticated Arduino programs will often have very in-depth structures that are not only defined but complemented by object-oriented programming paradigms. In essence, object-oriented programming is the idea that all things in programming should be abstracted and modular.

Essentially, things should be moving towards readability and reusability. Additionally, a huge focus should be placed on the ability to maintain code over a long period of time and easily make changes to it without having to go into long refactoring cycles.

So, moving on, there are some key object-oriented concepts that many books tend to gloss over. We're going to focus on these more in-depth in a logical sense here.

The first is *overloading*. Overloading is a major factor in object-oriented programming, but it's usually treated in passing or as something to learn "on the job." Overloading is the concept of things being made multi-purpose. That's it. While this doesn't seem incredibly detailed or articulate, there's not much more to it than this.

There are a few different forms of overloading, but the most prevalent is *function overloading*. Function overloading is when you give more than one definition of a function to the same name, which allows the function to execute in different manners dependent upon the arguments given.

There also is *overriding*. Overriding is another form of overloading. It's a very important concept, though. Before we discuss overriding, we should stop to be sure that you understand what *inheritance* is.

We assume that you know what classes are; however, it's worth noting that classes can *come from* classes. This is called inheritance. What happens is that a new child class is defined, and this class inherits the various methods and variables from its parent class, in addition to any newly defined methods and variables.

Often, in these cases, it will be necessary to redefine a function from a parent class. In this case, one should use function *overriding*. This is when you use the override keyword with the same function and arguments to essentially tell the program that this object needs to use its *own* function instead of its parent's version of the function.

There’s one more object-oriented concept that many books neglect to cover, and that’s the issue of the object-oriented *ethos*; not every program will require object-oriented programming. However, with how much it’s talked about, people often make it seem like object-oriented programming is the end-all-be-all of programming. You shouldn’t go around defining classes when there isn’t any need to.

However, chances are that there *will* be a need to at some point for you as a programmer, especially when you start working with really complex Arduino programs. In these cases, remember that the object-oriented ethos revolves around readability and modularity. At no point should a class definition or something of the like make your code more difficult to understand. It should be easy to add to or remove from, and it should be easy to change at a moment’s notice.

With that, we’ve covered a lot of the inherent logic behind programming and how all of it applies to Arduino programming. In the coming chapters, we’re going to be discussing further methods that will allow you to take your Arduino programming experience to the next level.

Chapter 4: Spend Time Thinking Outside the Box (and the Arduino)

In the previous chapters, we've spent a lot of time talking about programming concepts and how they specifically apply to Arduino programming. In this chapter, we're going to talk about how creative thinking and spending time away from Arduino programming both might enable you to be a better Arduino programmer in the end.

This may not make a lot of sense, but the fact is that both of these ideas have a lot of credence. No masterful painter ever got to their mastery by working simply with oils, though they may strongly *prefer* to work with oils. The same applies here: without thinking outside the box and working outside of your comfort zone, you're going to be missing out on a lot of different things that would lead you to become a better programmer and a better Arduino builder in general.

Here's a simple fact for you: all of creativity is based on the way that the brain processes inputs and outputs. Although there is some variety of spontaneous components, you can only ever invent things which are based on those things you've already learned or those things you've already been made aware of. Your ideas will almost always be based on those inputs that you've already dealt with in the past instead of being based on things that are spontaneously generated in terms of new ideas.

Spontaneous generation of ideas can lead to really cool and abstract things, but even the most spontaneous ideas are based on learned stimuli, from a psychological perspective. In other words, no thought that you’ve ever had has been completely original, because all of your thoughts are formed by the world around you and the unique way in which you happen to process all of that information.

If you want to be a good Arduino programmer, you really need to not stop at Arduino programming. For example, the fundamental languages which power the Arduino language itself, C and C++, have been used for a massive variety of different utilities in the nearly forty years that they’ve been around. At some point in these languages’ long histories, there has been something done that you’ve never even thought of, surely.

It is by taking in these inputs and getting this practice that you enable yourself to become a better programmer and do more.

I think that a good example would harken back to the phenomenon known as a Magic Mirror. Magic Mirrors are computer monitors which display information fed in by a Raspberry Pi, a microcomputer not too different from an Arduino through a tad bit more powerful and intended for different purposes. The monitor is tucked behind a one-sided mirror such that the information on the monitor displays on the reflective side of the mirror.

What results is a mirror that is incredibly science fiction becoming a science reality.

The thing is that while the project itself is simple, the logic behind the program really isn’t; for example, to display the data, a modified version of the Chromium browser is used. Within that Chromium browser, a custom page is built using HTML, CSS, and JavaScript (to retrieve information from the web).

It is through the knowledge of how to make these different things happen that a project as ambitious as the Magic Mirror was made possible. If one didn’t have the knowledge of how to modify and recompile the Chromium browser, nor if they didn’t have the knowledge on how to build web pages using HTML, CSS, and JavaScript, the project simply would have never happened; it would have remained science fiction.

In other words, you may have incredible ideas, but actually being able to take action to make them happen is an entirely different beast altogether. One thing is for certain: building up the knowledge required to, for example, build web pages or modify and recompile a web browser then write a bash script to automatically launch it upon the operating system starting up are things which go well beyond the scope of hobbyist programming for the Arduino.

You can ask for the help of other people, but that will only get you so far and nudge you along a little at a time. So what’s the other option, then?

All of this chapter has been building up to this: your ability to program Arduino sketches and to make your dream projects happen is based on you, not programming in Arduino at all.

Programming is a common set of skills that usually transfer across projects, but as an Arduino programmer, you generally are not building these skills in the optimal way. Reading books like this one is a start because they teach you the underlying concepts to all of the computer science mumbo-jumbo that you’re being fed, which hopefully acts to help make things click a little bit. But there are a lot of bigger concepts that you aren’t going to pick up just within your Arduino IDE. These are things like working with APIs, learning how to read documentation, learning how to create header files or libraries to make your programs more modular, learning in general how to be a better programmer.

These are skills which you gain through continually challenging yourself and trying to come up with new things.

And this has a little perk tacked onto the end of it, too: if you try to work on new things, you will be inspired more often. You’ll learn how to do things and start having ideas that you wouldn’t have had otherwise, because you’ll be enjoying new experiences and getting all sorts of new inputs. When you think about things you’ve never thought about before, your brain interprets this as a good thing - as a learning experience. This will expand your mind and make you more creative, in turn making you simultaneously a better programmer and a better tinkerer.

So in other words, if you want to be a good Arduino programmer, one of the best things that you can possibly do for yourself is to start working on projects that aren’t Arduino-based. You need to be exposing yourself to new things and challenging yourself to become better all of the time.

Chapter 5: Learn the Implementation of Algorithms

In this chapter, we’re going to discuss the implementation of algorithms in Arduino programming. Often, algorithms are understated in terms of their importance to Arduino programming, but using them, you’re able to do many things that you wouldn’t be able to otherwise.

An algorithm is, for lack of a better term, a way of standardizing a sort of procedure. We’re going to be discussing two different types of algorithms and how they relate to Arduino programming, as well as discuss in both a theoretical and a practical sense how they can be implemented.

We first will look at the bubble sort algorithm and discuss sorting in relation to Arduino programming. Bubble sorting is the most simple form of sorting, but it will give you a decent look into algorithmic programming and a greater example of how algorithms actually are in practice.

The second thing that we will look at is a Bayesian probability algorithm which is important in statistical programming. Bayesian probability is also used to determine “true probability,” which is probability that takes into account false positives and false negatives. We’re going to be discussing why probability algorithms may be useful in Arduino programming and how you may find yourself using them in the future depending upon your various different projects.

Let’s start by looking at sorting and thinking about when it could be useful to us as Arduino programmers. Sorting is the idea of taking some set of data and then filtering through it and moving things around accordingly. Let’s say, for example, that we had an array of ten values. There are times where we may need to sort these. While there do exist certain functions in the Arduino library for getting the minimum or maximum of two values, there is no built-in sorting function for an array, and there’s no built-in way to obtain the largest and smallest values within an array.

This is an introduction to algorithmic thinking more than anything else because Arduino is a perfect proxy to building bigger and better technological systems based off of concepts such as artificial intelligence and the internet of things. More than that, algorithms come up in complex programs and having some idea as to how to break down what an algorithm needs to do and then implementing it is a first important step to developing some sort of methodology for programming algorithmically.

The sample language for this example will be C/C++, but the key concepts will remain the same across any language. This is the first time in this book that we're not going to be using pseudocode.

Let's first define an array of 10 random values. You can make them whatever. Here are mine:

```
int numbers[10] = { 39, 63, 10, 70, 23, 34, 63, 13, 76, 34 };
```

This is the first important step. Afterward, we need to define the how our algorithm will work. Let's think about this for a second.

What a bubble sort essentially does is look at any given value in an array and compare it to a number either immediately before or after it. If the number before or after is larger or smaller (implementation can vary), then the two numbers will be swapped.

This might seem straightforward: you simply iterate through the array and perform checks to see if a given element is larger than another, right?

Not quite. If you used a single loop to iterate through the array, you're not actually going to be accomplishing much of anything. Instead, you need to use two for loops. The first loop will denote what we can call our *active position*. This will move through every element in the array one by one and perform the necessary checks and swaps until an *n*th element has been checked.

After that, you'll use a second for loop. This creates an *active integer* and compares it against every other element in the array. This is the loop of action. The other is the loop of iteration.

Within the loop of action, we must, therefore, create some kind of check mechanism. We can do this by comparing the active integer against another element in the array. If the other element is smaller, the two elements will swap positions. This means that the smaller element will be pushed towards the front of the array.

By now, we can assume that this means we need two functions: a *sort* function which will contain the logic of our sort mechanism, and a *swap* function which can perform the swap of the two integers given their addresses in memory.

We can implement these two like so:

For the *swap* function, we're going to want it to take the argument of two pointer addresses. Within the function, it will then define a temporary placeholder variable which we can call *i*. The placeholder will be used to store the value of integer 1. Integer 1 will then assume the value of integer 2. Integer 2 then assumes the value of the placeholder, which is the old value of integer 1, meaning integer 1 and 2 have now effectively swapped positions. Here is how I would define this function:

```
void swap(int *p1, int *p2)
{
        int i = *p1;
        *p1 = *p2;
        *p2 = i;
}
```

Easy peasy. The logic behind it is a little rough, but that's all an important part of the learning process!

Now, moving on beyond this, we're going to now work with our *sort* function. The sort function will create two iterative arrays. It will accept the arguments of the array's size and the array itself. It will iterate through these accordingly. Since we're comparing in a forward manner, the active position should only extend to the size of the array minus one element;

otherwise, when the last element in the array is reached, there will be an element overflow error, and the program will crash (if it executes at all.)

For the second for loop, we need to iterate according to the size minus the active position minus 1.

Within the second for loop, we need to define an if statement which checks to see whether the active integer is larger than the number immediately ahead of it. If it is, then the two swap positions, meaning we throw the memory addresses of the two to the swap function.

Here is how I would define this function:

```
void sort(int myArray[], int size)
{
        for (int i = 0; i < size - 1; i++)
        for (int j = 0; j < size - i - 1; j++)
        if  (myArray[j] > myArray[j+1])
        swap(&myArray[j], &myArray[j+1]);
}
```

Again, it’s not a terribly difficult algorithm, but it does require a bit of thought and is a decent introduction to algorithmic thinking if you’ve never done so before. To test this all, we can create a program with our test array that will do all of this, then print it out for us so that we can see if all is in working order:

```
#include <stdio.h>

void swap(int *p1, int *p2)
{
        int i = *p1;
        *p1 = *p2;
        *p2 = i;
}

void sort(int myArray[], int size)
{
        for (int i = 0; i < size - 1; i++)
        for (int j = 0; j < size - i - 1; j++)
        if  (myArray[j] > myArray[j+1])
        swap(&myArray[j], &myArray[j+1]);
}

int main()
{
        int numbers[10] = { 39, 63, 10, 70, 23, 34, 63, 13, 76, 34 };
        int size = sizeof(numbers)/sizeof(numbers[0]);
        sort(numbers, size);

        for (int i = 0; i < size; i++)
        {
        printf("%d", numbers[i]);
        }
}
```

Your end program should look somewhat like the one above. If so, then you've succeeded.

The next thing that we're going to discuss in an algorithmic sense is probability using a very rudimentary version of Bayes' theorem.

So what exactly is Bayes' theorem, and why is it useful as an Arduino programmer? Well, Bayes' theorem in and of itself is a way of determining true probability of a given situation based upon the likelihood that something has happened in the past. It takes into account various given rates and then returns a certain dimension based on those rates.

For this, we can take two events: event X and event Y. We then have P(X) which is the likelihood of X, and P(Y) which is the likelihood of Y. P(X|Y) is the likelihood of event *X* given that *Y* is true, and P(Y|X) is the likelihood of event Y if event X is true.

Let's say we're trying to find the likelihood of event X given that Y is true, and we have data related to event Y is X is true. Bayes' theorem then allows us to look at it like so:

$$P(X|Y) = P(Y|X)*P(X) / P(Y)$$

In other words, the probability of event X given that Y is true is equivalent to the probability of Y given that X is true multiplied by the probability of X, then all of this divided over the probability of Y.

Bayes' theorem is a little difficult to grasp if you aren't looking at it in an intuitive manner. Take, for example, spam filtering, which is a rather common application of Bayes' theorem in the world of computer science.

Let's say that event X is the likelihood that the message is spam, and event Y is the likelihood that it contains certain words flagged as spam. P(X|Y) is the probability that it is spam given that it contains words flagged as spam. P(Y|X) is the probability that a certain word flagged as spam will be in a spam message. P(X) is the probability that any given message is spam, and P(Y) is the probability that the words within are flagged for spam.

Therefore, we can render the equation something like this:

The probability that a message is spam based on flagged words is equivalent to *the probability that the flagged words are in a spam message* multiplied by *the probability that a message is spam*, all divided by *the probability that flagged words are spam*.

This yields:

P(spam|words) = P(words|spam)*P(spam) / P(words)

This could be rendered algorithmically rather simply, and it's a very rudimentary probability equation. However, this does give us a solid starting point. The equation, too, can be mostly given over without much thought given to the conversion process.

For this, we're going to need just one function which returns *x given y* based upon *y given x*, *y*, and *x*. We'll call this function *calc_prob*. We will use this function to return a double value which will be saved to a variable and printed out to the console.

The finished code would look something like this:

```
#include <stdio.h>

double calc_prob(double ygivenx, double x, double y)
{
	return (ygivenx * x) / y;
}

int main()
{
	double spam = 3100, words = 6888, wordsgivenspam = 7000;
	double spamgivenwords = calc_prob(wordsgivenspam, spam, words);
	printf("%.2f", spamgivenwords);
}
```

With that, you've written your second algorithm. Probability algorithms become massively useful when you need your Arduino programs to be able to act predictively. For example, you could use your Arduino to model certain situations or react accordingly. While it's not powerful enough for hardcore number crunching, it will definitely be powerful enough for basic probability equations. This, for example, is nothing too intensive. So long as it can retrieve information to form a dataset, you can use much of the information in this book to make highly reactive Arduino sketches that could, ideally, change the world.

The purpose of introducing these two algorithms was to give you insight as to how algorithms might impact your programming and how thinking algorithmically can be a major boon to your ability to program in the first place. Consider that the ability to mentally process and break apart certain functions is foundational to being able to think like a programmer, which - in the end - is what this book is trying to help you do.

Conclusion

Thanks for making it through to the end of *Arduino: Best Practices To Excel While Learning Arduino Programming*, let’s hope it was informative and able to provide you with all of the tools you need to achieve your goals whatever it may be.

The next step is to start using all of this information to better yourself as a programmer. Join communities and start researching projects that you think you would like to do. Progress now is within an arm’s reach - make it happen!

Finally, if you found this book useful in any way, a review on Amazon is always appreciated!