Learn to Program with Assembly

Foundational Learning for New Programmers

Jonathan Bartlett

Apress®

Learn to Program with Assembly

Foundational Learning for New Programmers

Jonathan Bartlett

Apress®

Learn to Program with Assembly: Foundational Learning for New Programmers

Jonathan Bartlett
Tulsa, OK, USA

ISBN-13 (pbk): 978-1-4842-7436-1 ISBN-13 (electronic): 978-1-4842-7437-8
https://doi.org/10.1007/978-1-4842-7437-8

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Sitraka Rakotoarivelo on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to Dale and Cindy Hanchey. Learning from their wisdom set me up for a career full of success.

Table of Contents

About the Author

Jonathan Bartlett is a software developer, researcher, and writer. His first book, *Programming from the Ground Up*, has been required reading in computer science programs from DeVry to Princeton. He has been the sole or lead author for eight books on topics ranging from computer programming to calculus. He is a Senior Software Research and Development Engineer for Specialized Bicycle Components with a focus on cross-team and cross-platform integration work.

About the Technical Reviewer

Paul Cohen joined Intel Corporation during the very early days of the x86 architecture, starting with the 8086, and retired from Intel after 26 years in sales/marketing/management. He is currently partnered with Douglas Technology Group, focusing on the creation of technology books on behalf of Intel and other corporations. Paul also teaches a class that transforms middle- and high-school students into real, confident entrepreneurs, in conjunction with the Young Entrepreneurs Academy (YEA), and is a traffic commissioner for the city of Beaverton, Oregon, and on the board of directors of multiple nonprofit organizations.

CHAPTER 1

Introduction

1.1 The Purpose of the Book

Have you ever wondered how your computer works? I mean, how it *really* works, underneath the hood? I've found that many people, including professional computer programmers, actually have no idea how computers operate at their most fundamental level.

You need to read this book whether or not you ever plan on writing assembly language code. If you plan on programming computers, you need to read this book in order to demystify the operation of your most basic tool—the processor itself. I've worked with a lot of programmers over the years. While you can do good work only knowing high-level languages, I have found that there is a glass ceiling of effectiveness that awaits programmers who haven't learned the machine's own language.

Learning assembly language is about learning how the processor itself thinks about your code. It is about gaining the mind of the machine. Even if you never use assembly language in practice, the depth of understanding you will receive by learning assembly language will make your time and effort worthwhile. You will understand at a more visceral level the various trade-offs that are made with different programming languages and why certain high-level operations may be faster than others and get an overall sense of what your computer is really doing.

Additionally, while the practical uses of assembly language are getting fewer and further between, there are still many places where assembly language knowledge is needed. Compiler writers, kernel developers, and high-performance library implementers all utilize assembly language to some degree and probably always will. Additionally, embedded developers, because of resource constraints, often program in assembly language as well.

J. Bartlett, *Learn to Program with Assembly*, https://doi.org/10.1007/978-1-4842-7437-8_1

1.2 Who Is This Book For?

This book is for programmers at any level. This book should work as your first or your fortieth programming book. Some later chapters will assume some familiarity with various programming languages, but the core content is written so that anyone can pick it up and read it.

I generally assume some working knowledge of Linux and the command line. However, if you haven't used the command line, Appendix B will give a brief introduction.

If you don't use Linux as your primary operating system, that's okay, too. I've built a Docker image that is customized to work with this book, and Appendix A will help you get started using it.

You only need to know the basics—how to run programs on the command line, how to edit text files, etc. If you have done any work at all on the command line (or have read and worked through Appendix B), you probably know everything that you need to get started. If you haven't, there are numerous tutorials on the Internet about getting started on the command line. You don't need to be an advanced systems administrator. If you know how to change location, edit files, and create directories, that's all the skills you actually need.

1.3 Why Learn Assembly Language?

In the modern age of modern programming languages where a single line of code can replace hundreds of lines of assembly language, why bother to study assembly language at all? The fact is assembly language is how your computer runs. Any good craftsman knows how their tools work, and computer programming is no different. Knowing your tools helps you get the most out of them.

The biggest advantage is one that is hard to point to concretely—it is simply understanding how the pieces fit together. Some people are perfectly happy not knowing how the tools that they work with actually function. However, those people often wind up being mystified by certain problems and then have to go to someone who actually knows how these tools function to figure it out. Knowing assembly language makes *you* the guru who understands how everything fits together.

Of course, there are also more practical reasons I can point to. Understanding how many security exploits work relies on understanding how the computer is

actually operating. So, if your goal is to do computer security work, in order to actually understand how hackers are manipulating the system, you have to know how the system works in general.

Some people learn assembly language so that they can make faster programs. While modern optimizing compilers are really great at making fast assembly language, since they are computer programs, they can only operate according to fixed rules and axioms. Human creativity, however, allows for the creation of new ideas which go beyond what computers are programmed to do.[1]

There are other cases where assembly language is actually simpler for programming. For many embedded processors and applications, programming in a high-level language is actually *harder* than just programming in assembly language directly. If you are doing low-level work with hardware working with individual bits and bytes, then assembly language oftentimes winds up being *more* straightforward and easy to program in than a high-level language.[2]

There are also many areas of modern programming on standard computers which *must* happen in assembly language, or at least require a background knowledge of it. Compilers, new programming languages, operating system code, drivers, and other system-level features all require either direct assembly language programming or a background knowledge of it.

Again, I will say that, for me, the greatest benefit of learning assembly language programming is simply gaining a better mental model for what is happening in the computer when I'm programming. When people describe security exploits, I can understand what they are talking about. When people describe why some programming feature "costs" too much in terms of execution speed, I have a mental framework to understand why. When low-level issues arise, I have a feel for what sorts of things might be causing problems.

[1] The optimal methodology is actually to combine both humans and computers and let the computer apply the fixed rules and let human creativity see where they can improve upon them.

[2] Note that most embedded processors will use a *different* assembly language than the one in this book. Nevertheless, I think that you will find learning the assembly language that is on your own computer beneficial and that most of the *ideas* transfer easily to other processors, even if the instructions are a little different. Embedded processors come with a whole host of their own difficulties, so having mastery of assembly language *in general* before trying to program an embedded processor is definitely worthwhile.

1.4 A Note to New Programmers

If you are reading this book and you are new to programming, I want to offer a special word to you. While I think you have made a good choice using this book to learn programming, I want you to know that it may not be as exciting as other programming languages. Reading this book will help you to gain the understanding of the processor to make you great at programming. Because you know all the things the computer is doing under the hood, you will have insights when doing more exciting types of programming that others won't have.

However, assembly language itself is not incredibly exciting to write. You are literally doing everything by hand, so even doing simple things tends to take a long time. The purpose of higher-level programming languages is to speed up the process of writing code. What I don't want you to do is to read this book and then think, "Oh my! Programming takes so much work!" Remember, most of us got into this business to automate things, and that includes automating the task of programming. Many experienced programmers can pack a lot of juice into even a single line of code in a high-level language.

If you don't know, programming languages are generally grouped into "high-level" and "low-level" languages. Higher-level languages are focused more on making code that matches more closely the problem you are trying to solve, while lower-level languages are focused on making code that more closely follows the computer's own mode of operation. Assembly language is the almost-lowest-level language there is. The instructions in assembly language exactly match the instructions that the processor executes. The only thing lower than assembly language is writing machine opcodes (see Appendix K if that is of interest to you). As you will see, computers translate *everything* into numbers. That includes your programs. However, it would be hard to read and manipulate a program if it were just numbers. Therefore, almost everyone writes the actual code in assembly language and then uses a program (called an assembler) to translate that into machine code. Assembly language is basically human-readable machine code.

That is why I say that learning assembly language will give you insight into the operation of the computer. Unlike other programming languages, when you learn assembly language, you are learning to program the computer on its own level. I've generally found that it is somewhat dangerous to automate a process you don't understand, especially for someone who is trying to be an expert. An expert mathematician will certainly use software to aid their thinking, but only because they

know what the software is automating. An expert race car driver will certainly use their car's steering system to maneuver, but they will still know how the car is operating underneath. This helps them understand how decisions they make at the wheel will affect various system components such as the tread on the tires or gasoline usage. As a casual driver, these things aren't important to me, so my understanding generally stops at the steering wheel and the gas tank. However, if I planned on being a performance race car driver, even if I never maintained the car myself, even if I had a whole crew that did that for me, I would still be well served to understand the car at its deepest level in order to get the most out of it at critical junctures.

Different people have different ideas, but, if you are willing, I definitely suggest starting with assembly language. It will cause you to think differently about problems and computers and ultimately will shape your thinking to more closely match what is required for effective computer programming.

1.5 Types of Assembly Language

Note that there is not a single type of machine language for all computers, although most PCs share the same machine language. Machine languages are usually divided up by **instruction set architecture** (**ISA**). The ISA refers to the set of instructions that are allowed by the computer. Many, many different computers share the same ISA, even when built by different manufacturers. Almost all modern PCs use the **x86-64** ISA (sometimes referred to as **AMD64**). Older PCs use the **x86** ISA (this is the 32-bit version of x86-64). Many cell phones use a variation of the **ARM** ISA. Finally, some older game consoles (and really old Macs) use the **PowerPC** ISA. Many other ISAs exist, but are usually restricted to chips that have very specialized uses, such as in embedded devices.

The ISA covered in this book is the x86-64 ISA. This was developed by AMD as a 64-bit extension to the 32-bit x86 ISA developed by Intel. It is now standard in PC-based systems and most servers.

In addition, since assembly language uses human-readable symbols that translate into machine code, different groups have implemented assembly language using different syntaxes. There is no difference in the final machine code, but the different syntaxes have different looks. The two main syntaxes are NASM syntax (sometimes called Intel syntax) and AT&T (sometimes called GAS) syntax. Again, there is no difference in functionality, only in look. We will use AT&T syntax here, because this is the syntax used both in the Linux kernel and as the default syntax by the GNU Compiler

Collection (GCC) toolchain. If you need to use NASM syntax for some reason, a quick translation guide between the two syntaxes is available in Appendix D.

Finally, different operating systems utilize the chips in different ways. The focus here will be on 64-bit Linux-based operating systems. You will need to be running a 64-bit Linux-based operating system to use this book. However, as noted, if you are not on Linux, you can use the Docker setup in Appendix A to run a compatible Linux instance inside a 64-bit Mac or a 64-bit PC.

1.6 Structure of This Book

This book is arranged into three basic parts. This chapter and the next are introductory material before the main parts of the book. They are here to get you started, but are not really about how to program in assembly language.

Part I of the book focuses on the basics of assembly language itself. The programs are not very exciting, because assembly language itself doesn't do much except move data around and process it. Because we are limiting ourselves to assembly language itself, the results of these programs are always numbers. However, the simple nature of the programs will help you get a good feel for assembly language and how it works before trying more complicated things such as input/output. New instructions will still be provided in subsequent parts of the book, but you should have a pretty good feel for assembly language by the time you finish this part of the book. Additionally, most of what you learn in this part is transferable to any other operating system running on a CPU with the x86-64 instruction set.

Part II of the book goes into detail on how programs interact with the operating system. This includes things like displaying to the screen, reading and writing files, and even a bit of user input. It also includes some system management features, such as how to interact with system libraries and how to request more memory from the operating system. This part is very specific to the Linux operating system. While most operating systems provide similar facilities, the specifics of how to use them are unique to the particular operating system you are using.

Part III of the book discusses how programming languages get implemented at the lowest level. Being an introductory book, the goal here isn't to teach you the *best* way to implement programming languages, but rather to give you a feel for the kinds of things that the computer is doing under the hood in various programming languages. How would someone implement feature X, Y, or Z? If modern programming languages amaze

and mystify you, Part III should help to make them less enigmatic. Part III is not about a particular programming language, but will guide you through various types of language features that you may find in any number of programming languages.

If this is your first book on computer programming, my recommendation is to stop after Part II and then come back and read Part III after you have gained some experience with other programming languages. This will provide the needed context for understanding Part III of the book.

Part IV of the book has several appendixes that cover various topics that are important to know, but don't quite fit anywhere within the main text. As you are interested, take a look at the appendixes to find short introductions to various topics.

The best way to learn programming is by doing. I would suggest programming every example written in the text yourself to make sure that you fully understand what is occurring. Additionally, every chapter ends with a list of exercises. Those exercises are intended to help you make practical use of what you know and give you experience in thinking about programming on the assembly language level.

CHAPTER 2

The Truth About Computers

I'm going to now share with you the shocking truth about computers—computers are really, really stupid. Many people get enamored with these devices and start to believe things about computers that just aren't true. They may see some amazing graphics, some fantastic data manipulation, and some outstanding artificial intelligence and assume that there is something amazing happening inside the computer. In truth, there *is* something amazing, but it isn't the intelligence of the computer.

2.1 What Computers Can Do

Computers can actually do very few things. Now, the modern computer instruction set is fairly rich, but even as the number of instructions that a computer knows increases in abundance, these are all primarily either (a) faster versions of something you could already do, (b) computer security related, or (c) hardware interface related. Ultimately, as far as computational power goes, all computers boil down to the same basic instructions.

In fact, one computer architecture, invented by Farhad Mavaddat and Behrooz Parham, only has one instruction, yet can still do any computation that any other computer can do.[1]

So what is it that computers can do computationally? Computers can

- Do basic integer arithmetic
- Do memory access

[1] For those curious, the instruction is "subtract and branch if negative." If you don't know what that means, it will make a lot more sense by the time you finish this book. If you want to know more about this computer, the paper is "URISC: The Ultimate Reduced Instruction Set Computer" in the *Journal of Electrical Engineering Education*, volume 25. These sorts of computers are known today as OISC systems ("one instruction set computers").

J. Bartlett, *Learn to Program with Assembly*, https://doi.org/10.1007/978-1-4842-7437-8_2

- Compare values
- Change the order of instruction execution based on a previous comparison

If computers are this limited, then how are they able to do the amazing things that they do? The reason that computers can accomplish such spectacular feats is that these limitations allow hardware makers to make the operations very fast. Most modern desktop computers can process over a *billion* instructions *every second*. Therefore, what programmers do is leverage this massive pipeline of computation in order to combine simplistic computations into a masterpiece.

However, at the end of the day, all that a computer is really doing is really fast arithmetic. In the movie *Short Circuit*, two of the main characters have this to say about computers—"It's a machine… It doesn't get happy. It doesn't get sad. It doesn't laugh at your jokes. It just runs programs." This is true of even the most advanced artificial intelligence. In fact, the failure to understand this concept lies at the core of the present misunderstanding about the present and future of artificial intelligence.[2]

2.2 Instructing a Computer

The key to programming is to learn to rethink problems in such simple terms that they can be expressed with simple arithmetic. It is like teaching someone to do a task, but they only understand the most literal, exact instructions and can only do arithmetic.

There is an old joke about an engineer whose wife told him to go to the store. She said, "Buy a gallon of milk. If they have eggs, get a dozen." The engineer returned with 12 gallons of milk. His wife asked, "Why 12 gallons?" The engineer responded, "They had eggs." The punchline of the joke is that the engineer had over-literalized his wife's statements. Obviously, she meant that he should get a dozen *eggs*, but that requires context to understand.

The same thing happens in computer programming. The computer *will* hyper-literalize every single thing you type. You must expect this. Most bugs in computer programs come from programmers not paying enough attention to the literal meaning of what they are asking the computer to do. The computer can't do anything except the literal meaning.

[2] For more information about this issue, see Erik Larson's book, *The Myth of Artificial Intelligence: Why Computers Can't Think the Way We Do*. I've also written about this some—see my article "Why I Doubt That AI Can Match the Human Mind," available at https://mindmatters.ai/2019/02/why-i-doubt-that-ai-can-match-the-human-mind/.

Learning to program in assembly is helpful because it is more obvious to the programmer the hyper-literalness of how the computer will interpret the program. Nonetheless, when tracking down bugs in any program, the most important thing to do is to track what the code is actually saying, not what we meant by it.

Similarly, when programming, the programmer has to specify *all* of the possible contingencies, how to check for them, and what should be done about them. Imagine we were programming a robot to shop for us. Let us say that we gave it the following program:

1. Go to the store.
2. If the store has corn, buy the corn and return home.
3. If the store doesn't have corn, choose a store that you haven't visited yet and repeat the process.

That sounds pretty specific. The problem is, what happens if no one has corn? We haven't specified to the robot any other way to finish the process. Therefore, if there was a corn famine or a corn recall, the robot will continue searching for a new store *forever* (or until it runs out of electricity).

When doing low-level programming, the consequences that you have to prepare for multiply. If you want to open a file, what happens if the file isn't there? What happens if the file is there, but you don't have access to it? What if you can read it but can't write to it? What if the file is across a network, and there is a network failure while trying to read it?

The computer will only do exactly what you tell it to. Nothing more, nothing less. That proposition is equally freeing and terrifying. The computer doesn't know or care if you programmed it correctly, but will simply do what you actually told it to do.

2.3 Basic Computer Organization

Before we go further, I want to be sure you have a basic awareness of how a computer is organized conceptually. Computers consist of the following basic parts:

- The CPU (also referred to as the processor or microprocessor)
- Working memory
- Permanent storage
- Peripherals
- System bus

Let's look at each of these in turn.

The **CPU** (central processing unit) is the computational workhorse of your computer. The CPU itself is divided into components, but we will deal with that in Section 2.7. The CPU handles all computation and essentially coordinates all of the tasks that occur in a computer. Many computers have more than one CPU, or they have one CPU that has multiple "cores," each of which is more or less acting like a distinct CPU. Additionally, each core may be hyperthreaded, which means the core itself to some extent acts as more than one core. The **permanent storage** is your hard drive(s), whether internal or external, plus USB sticks, or whatever else you store files on. This is distinct from the **working memory**, which is usually referred to as **RAM**, which stands for "random access memory."[3] The working memory is usually wiped out when the computer gets turned off.

Everything else connected to your computer gets classified as a **peripheral**. Technically, permanent storage devices are peripherals, too, but they are sufficiently foundational to how computers work I treated them as their own category. Peripherals are how the computer communicates with the world. This includes the graphics card, which transmits data to the screen; the network card, which transmits data across the network; the sound card, which translates data into sound waves; the keyboard and mouse, which allow you to send input to the computer; etc.

Everything that is connected to the CPU connects through a **bus**, or **system bus**. Buses handle communication between the various components of the computer, usually between the CPU and other peripherals and between the CPU and main memory. The speed and engineering of the various computer buses is actually critical to the computer's performance, but their operation is sufficiently technical and behind the scenes that most people don't think about it. The main memory often gets its own bus (known as the front-side bus) to make sure that communication is fast and unhindered.

Physically, most of these components are present on a computer's motherboard, which is the big board inside your desktop or laptop. The motherboard often has other functions as well, such as controlling fans, interfacing with the power button, etc.

[3] It's called random access memory because you can easily access any given part of the memory. This was in comparison to disks or tape, in which you had to physically move the read/write head to the right spot before you could read the data. Modern solid state drives are essentially random access as well, but we still use the term RAM to refer to the main memory, not the disks.

2.4 How Computers See Data

As mentioned in the introduction, computers translate everything into numbers. To understand why, remember that computers are just electronic devices. That is, everything that happens in a computer is ultimately reducible to the flow of electricity. In order to make that happen, engineers had to come up with a way to represent things with flows of electricity.

What they came up with is to have different voltages represent different symbols. Now, you could do this in a lot of ways. You could have 1 volt represent the number 1, 2 volts represent the number 2, etc. However, devices have a fixed voltage, so we would have to decide ahead of time how many digits we want to allow on the signal and be sure sufficient voltage is available.

To simplify things, engineers ultimately decided to only make two symbols. These can be thought of as "on" (voltage present) and "off" (no voltage present), "true" and "false," or "1" and "0." Limiting to just two symbols greatly simplifies the task of engineering computers.

You may be wondering how these limited symbols add up to all the things we store in computers. First, let's start with ordinary numbers. You may be thinking, if you only have "0" and "1," how will we represent numbers with other digits, like 23? The interesting thing is that you can build numbers with any number of digits. We use ten digits (0–9), but we didn't have to. The Ndom language uses six digits. Some use as many as 27.

Since the computer uses two digits, the system is known as **binary**. Each digit in the binary system is called a **bit**, which simply means "binary digit." To understand how to count in binary, let's think a little about how we count in our own system, **decimal**. We start with 0, and then we progress through each symbol until we hit the end of our list of symbols (i.e., 9). Then what happens? The next digit to the left increments by one, and the ones place goes back to zero. As we continue counting, we increment the rightmost digit over and over, and, when it goes past the last symbol, we keep flipping it back to zero and incrementing the next one to the left. If that one flips, we again increment the one to the left of that digit, and so forth.

Counting in binary is exactly the same, except we just run up against the end of our symbol list much more quickly. It starts at 0, then goes to 1, and then, hey, we are at the end of our symbols! So that means that the number to the left gets incremented (there is always imaginary zeroes to the left of the digits we have) and our rightmost digit flips

back to zero. So that means that after 0 and 1 is 10! So, counting in binary looks like this (the numbers on the left are the equivalent decimal numbers):

0. 0
1. 1
2. 10 (we overflowed the ones position, so we increment the next digit to the left and the ones position starts over at zero)
3. 11
4. 100 (we overflowed the ones position, so we increment the next digit to the left, but that flips that one to zero, so we increment the next one over)
5. 101
6. 110
7. 111
8. 1000
9. 1001
10. 1010
11. 1011
12. 1100

As you can see, the *procedure* is the same. We are just working with fewer symbols.

Now, in computing, these values have to be stored somewhere. And, while in our imagination, we can imagine any number of zeroes to the left (and therefore our system can accommodate an infinite number of values), in physical computers, all of these numbers have to be stored in circuits somewhere. Therefore, the computer engineers group together bits into fixed sizes.

A **byte** is a grouping of 8 bits together. A byte can store a number between 0 and 255. Why 255? Because that is the value of 8 bits all set to "1": `11111111`.

Single bytes are pretty limiting. However, for historic reasons, this is the way that computers are organized, at least conceptually. When we talk about how many gigabytes of RAM a computer has, we are asking how many billions (giga-) of bytes (groups of 8 bits together) the computer has in its working memory (which is what **RAM** is).

Most computers, however, fundamentally use larger groupings. When we talk about a 32-bit or a 64-bit computer, we are talking about how the number of bits that the computer naturally groups together when dealing with numbers. A 64-bit computer, then, can naturally handle numbers as large as 64 bits. This is a number between 0 and 18,446,744,073,709,551,615.

Now, ultimately, you can choose any size of number you want. You can have bigger numbers, but, generally, the processor is not predisposed to working with the numbers in that way. What it means to have a 64-bit computer is that the computer can, in a single instruction, add together two 64-bit numbers. You can still add 64-bit numbers with a 32-bit or even an 8-bit computer; it just takes more instructions. For instance, on a 32-bit computer, you could split the 64-bit number up into two pieces. You then add the rightmost 32 bits and then add the leftmost 32 bits (and account for any carrying between them).

Note that even though computers store numbers as bits, we rarely refer to the numbers in binary form unless we have a specific reason. However, knowing that they are bits arranged into bytes (or larger groupings) helps us understand certain limitations of computers. Oftentimes, you will find values in computing that are restricted to the values 0–255. If you see this happen, you can think, "Oh, that probably means they are storing the value in a single byte."

2.5 It's Not What You Have, It's How You Use It

So, hopefully by now you see how computers store numbers. But don't computers store all sorts of other types of data, too? Aren't computers storing and processing words, images, sounds, and, for that matter, negative or even non-integer numbers?

This is true, but it is storing all of these things *as numbers*. For instance, to store letters, the letters are actually converted into numbers using ASCII (American Standard Code for Information Interchange) or Unicode codes (which we will discuss more later). Each character gets a value, and words are stored as consecutive values.

Images are also values. Each pixel on your screen is represented by a number indicating the color to display. Sound waves are stored as a series of numbers.

So how does the computer know which numbers are which? Fundamentally, the *computer* doesn't. All of these values look exactly the same when stored in your computer—they are just numbers.

What makes them letters or numbers or images or sounds is how they are *used*. If I send a number to the graphics card, then it is a color. If I add two numbers, then they are numbers. If I store what you type, then those numbers are letters. If I send a number to the speaker, then it is a sound. It is the burden of the programmer to keep track of which numbers mean which things and to treat them accordingly.

This is why files have extensions like `.docx`, `.png`, `.mov`, or `.xlsx`. These extensions tell the computer how to interpret what is in the file. These files are themselves just long strings of numbers. Programs simply read the filename, look at the extension, and use that to know how to use the numbers stored inside.

There's nothing preventing someone from writing a program that takes a word processing file and treating the numbers as pixel colors and sending them to the screen (it usually looks like static) or sending them to the speakers (it usually sounds like static or buzzing). But, ultimately, what makes computer programs useful is that they recognize how the numbers are organized and treat them in an appropriate manner.

If this sounds complicated, don't worry about it. We will start off with very simple examples in the next chapter.

What's even more amazing, though, is that the computer's instructions are themselves just numbers as well. This is why your computer's memory can be used to store both your files and your programs. Both are just special sequences of numbers, so we can store them all using the same type of hardware. Just like the numbers in the file are written in a way that our software can interpret them, the numbers in our programs are written in a special way so that the computer hardware can interpret them properly.

2.6 Referring to Memory

Since a computer has billions of bytes of memory (or more), how do we figure out *which* specific piece of memory we are referring to? This is a harder question than it sounds like. For the moment, I will give you a simplified understanding which we will build upon later on.

Have you ever been to a post office and seen an array of post-office boxes? Or been to a bank and seen a whole wall of safety deposit boxes? What do they look like?

Usually, each box is the same size, and each one has a number on it. These numbers are arranged sequentially. Therefore, box 2345 is right next to box 2344. I can easily find any box by knowing the number on the outside of the box.