Steven Frank

HOW TO COUNT

Programming for Mere Mortals, Vol. 1

How to Count

By Steven Frank

Book 1 of the "Programming for Mere Mortals" series

2nd Edition

Cover design & illustration by Neven Mrgan.

For Eliza

Acknowledgements

Thank you to the many authors, teachers, friends, and colleagues who've shaped me into the more-or-less competent programmer I am today. You have all made this book possible.

Thanks also to my co-conspirators and proofreaders who submitted feedback on my early drafts: David Grant, Dave Hayden, Andrew Kohan, Neven Mrgan, Alex Pasco, John Roquemore, and any others I may have inadvertently overlooked!

Last but certainly not least, thank you to my wife Annie and daughter Eliza for their patience while I was busy writing.

Goal of this book

There exist a tremendous number of books in the world dedicated to the topics of computers and programming. It's very common for them to focus on a particular programming language, operating system, or tool. There's nothing wrong with that.

What I've felt for some time has been missing is a more gentle, plain English introduction to some of the core concepts of programming for non-programmers.

There are recurring patterns and themes that are common to programming across all the computers and operating systems ever made, from Macs to Windows PCs to iPhones to graphing calculators to coffee machines and beyond. If I can help you master these central tenets, you'll be a better programmer regardless of what programming language or platform you end up working on.

That's what I hope to achieve with the "Programming for Mere Mortals" series, of which this is the first book: "How to Count".

How to count? Sounds a bit simplistic. Don't you already know how to count?

You do, but humans and computers count differently. I'll show you how a computer does it, and how to apply your knowledge of human counting to computer counting.

Like many subjects, understanding programming requires stacking layers of knowledge. It's easier to express a new concept in terms of an existing, familiar concept: "It's like *x* but with *y*." If I were going to teach you the Objective-C programming language, for example, we could save hours, possibly days, if you already knew the C programming language. But if you didn't, there'd be a lot more groundwork to cover.

If you go and buy a book on a specific programming language, you've skipped up a few layers in the stack. You might get comfortable with the language, but you won't truly understand what is happening inside the computer when you speak it.

It's like buying a Spanish phrasebook and learning that "¿Cómo te llamas?" is Spanish for "What's your name?". You'll come away knowing that particular phrase, but you won't understand how it's constructed, which word is the verb, how its grammar differs from English, or how you might re-combine the words in different ways to make other phrases that a Spanish speaker would understand.

Unfamiliarity breeds fear, and familiarity quashes it. People new to computers are often scared of them because computer systems have become vast, complex and brittle over the decades. Being familiar with programming, and by extension, understanding what is actually going on inside the box — even if you don't aspire to become a programmer yourself — helps with

this.

It's my belief that people who understand how computers work, even at an elementary level, will have an easier time down the road troubleshooting simple problems. People with barebones programming experience can create custom-made software tools to address their specific computing needs, without always being required to purchase and learn a particular application or tool to provide the solution for them.

Not everyone who buys a car needs to know how a piston engine works, but even a passing familiarity gives you insights that might help you solve a problem one day, or at least discuss the matter more competently with others.

I hope to teach you some of the core fundamentals of how programmers think and how computers work, in a very conversational, informal way.

My qualifications

To give you a bit of background about myself, and why you should listen to anything I have to say about computers: I am the co-founder of a small software company named Panic, Inc. Panic produces software for Mac computers, iPhone, and iPad. We wrote Audion, one of the first MP3 jukebox applications for the Mac, before iTunes existed.

We've been in business since 1997, slowly but steadily growing from two people to over a dozen. Our humble company has won many accolades including, more than once, the coveted Apple Design Award.

As for myself, I was fortunate enough to be born just a few years prior to the introduction of the Apple II, widely regarded as the first viable off-the-shelf personal computer. These were very simple computers compared to the ones we keep in our homes and offices today. I consider myself lucky to have been able to learn the basics when all there was to learn was the basics.

As I grew, so did the industry. Personal computers sprouted graphical windowed interfaces, modems, and eventually internet connectivity. I had no choice but to learn the stack of knowledge layer by layer starting from the bottom, simply because it grew up around me.

Thanks to that ground floor understanding, it was easier for me to place new concepts in orbit around what

I already knew than it would be for someone who hadn't had the same run-up.

My hope is this experience gives me a somewhat unique perspective to pass along this core knowledge to you. You can learn about computers and programming from the bottom up, like I did.

A note on math

There's a widely believed myth that you must be an expert in mathematics to be a programmer. Although it doesn't hurt, it's not strictly true. It's true for very high-level computer science topics. Comparatively few programmers exist in this realm. Those who do tend to solve these very high level problems for us and pass the solutions down in the form of chunks of ready-to-use program code.

You can, in my opinion, be a pretty effective programmer without much more than a grasp of basic algebra. Among my nerdy high school friends, I was the only one who scored higher (significantly) on the verbal portion of my SAT than the math portion. But yet, here I am making a living as a programmer.

There will be a little math — sorry, it can't be helped — but I'll do my best to break it down into easily digestible chunks for folks like me who do their best thinking in words, not numbers.

If you need to contact me with a question or feedback about the book, please email me. My address is stevenf@panic.com. On Twitter, I'm @stevenf.

Welcome, and enjoy the book!

Decimal

Whether you know it or not, you count using a particular and specific counting system called “decimal”, or more technically “base 10”.

What’s a “base”? It simply refers to the number of symbols available to you while counting. As the name implies, base 10 has ten of them. In order, they are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Perhaps these symbols look familiar to you? It doesn’t really matter how the symbols look. They could look like hieroglyphics, or the dots on dominos. It only matters that there are ten of them, and they’re all unique.

Why do humans count in base 10? A popular theory is that it’s easy for us to learn, since most of us have ten fingers on which to count. It’s possible that if humans were all born with six fingers, we’d have adopted base 6 instead. But we’re born with 10, usually, so here we are with our 10 symbols.

The most important thing to observe about base 10, and the key to understanding all other numerical bases, is what happens when you run out of symbols. Our 10 symbols by themselves get us up to the value nine, but what do we do when we need to represent the value ten? You do it so automatically, you probably don’t even think about it anymore:

You add one to the next digit to the left, and the 9 digit wraps around back to zero.

Think about a traditional mechanical odometer in an older car — the gauge that tracks your mileage. It has those rotating cylinders with the digits 0-9 on them. The odometer starts out showing 0 for each digit. As you drive the car, the rightmost digit goes up by 1 for each mile. When your mileage goes from nine to ten, it has to roll the number in the tens place from 0 to 1, and the number in the ones place rolls over from 9 back around to 0.

So, let's count in decimal! Starting at zero, here we go:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

The 9 wraps around to zero, and the (not previously shown) digit to the left increases by one:

10, 11, 12, 13, 14, 15, 16, 17, 18, 19

The 9 wraps around to zero, and the digit to the left increases by one:

20, 21, 22…

You just keep going like this until you hit 99, and need another place for a digit. But the rules don't change. The digit to the left increases by one, and the 9s wrap around to zero:

98, 99, 100, 101, 102…

And that's all you need to know to count as high as you like in decimal. Or any other numeric base, as we'll see next.

Binary

What if instead of 10 symbols, we only had 2? The good news is we'd still be able to represent any numeric value, just as we can in decimal.

It turns out there is indeed such a thing as base 2. It's called "binary", and it's very important to computers and programming.

The rules work exactly as they do in base 10, except our only two available symbols are:

0, 1

So, let's count in binary, starting again from the value zero:

0

1

Uh oh. We haven't even made it to the value two, and we've run out of symbols. What would we do in decimal? Think back to the odometer example. How would an odometer work, if each digit on the cylinders only went from 0 to 1 instead of 0 to 9? Well, when we needed to go past 1, we'd add 1 the digit to the left and wrap the rightmost digit around back to 0:

10

It looks like a ten, because we're used to decimal. But that's the decimal value two expressed in binary. Let's keep counting:

11

There's the decimal value three, but we've run out of symbols again. Same old odometer story: increase the digit to the left by 1, and wrap the 1s back around to 0:

100

101

110

111

Can you guess the next number?

1000

The binary number 1000 is the same value as the decimal number 8. It looks like "one thousand" to us, because we want to think in decimal, and it just so happens that both decimal and binary use two of the same symbols: 0 and 1. But you can plainly see that we've only counted 8 numbers, not a thousand.

Why is binary so important to computers? Because a single binary digit — or "bit" for short — is the smallest

individual unit of data that a computer can deal with.

Bits are the “atoms” of everything digital. Every piece of information stored or processed by a computer is made up of some number of bits. Sometimes just a few, sometimes thousands or millions. But everything the computer understands is, at its lowest level, made up of bits. Text, graphics, sounds, program instructions — all made of bits in one way or another. I’ll explain how in a later volume.

Why do computers want to speak in binary? Because bit values can be represented very easily using electricity within the computer’s circuitry. The presence of electricity on a wire might indicate a “1”. The absence of electricity, a “0”. Just like a light switch, you can represent one of two states, on or off, by whether or not electricity is coming down a wire.

Now imagine you had a few million light switches, each attached to a light. How many possible combinations of lights off and on could you have? Quite a few. In fact, for each additional light switch you add, you double the number of possible combinations. In just a moment, I’ll show you how that works.

You may have heard the term “8-bit” in reference to very early personal computers. To what does this refer? It means the processors in those vintage machines were able to most efficiently work with units of data 8 bits in size.

Here's an arbitrary 8 bit binary number:

0001 1011

I've split it into two groups of four digits to make it easier to read, but you should consider it eight contiguous bits. How could we figure out what the decimal value of that binary number is?

You might not have noticed, but as we were doing our binary counting exercise, each time we had to add another digit to the left, we'd gone up by a power of two in decimal:

Binary	Decimal	Power of Two
1	1	2^0
10	2	2^1
100	4	2^2
1000	8	2^3
10000	16	2^4
100000	32	2^5
1000000	64	2^6
10000000	128	2^7

(What does "power of two" mean? It's a math term, meaning to multiply the number 2 by itself a certain number of times. In other words, 2^3 – read as "two to the power of three" or "two to the third power" – means "multiply two by itself three times", or put another way: $2 \times 2 \times 2$. There's a special exception for 2^0: two to the power of 0 equals 1.)

If you look at the previous chart you can easily see that every time you add another bit to a binary number, you've doubled the range of possible numeric values that can be represented.

With just 8 bits, you can represent 256 unique numeric values, starting with 0 and counting up to 255. In modern-day computers, a group of 8 bits is also called

a “byte”.

Also notice in the chart that when a power of two is represented in binary, it is simply a “1” followed by that many “0”s. For example, the decimal number 8 is equal to 2^3, and indeed the binary representation is a “1” bit followed by three “0” bits.

If you’re visually inclined, like me, imagine our mystery 8 bit number as a series of 8 boxes, each box having a corresponding decimal value. Then, for each box that contains a “1” bit, add up the corresponding decimal numbers:

Decimal:	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Binary:	0	0	0	1	1	0	1	1

Our binary number has "1"s at the decimal 16, 8, 2, and 1 positions. Add those together:

16 + 8 + 2 + 1

= 27

The 8-bit binary number 00011011 is equal to the decimal number 27.

How would we go the other way, from decimal to binary? What would be the binary representation of the decimal number, oh, let's say 53?

We just need to know which powers of two add up to 53. You can figure this out by starting with the largest power of two that's less than or equal to 53, subtracting it, then repeating the process until you reach zero:

53 – 32 = 21

21 – 16 = 5

5 – 4 = 1

1 – 1 = 0

We've shown above that 53 is equal to 32 + 16 + 4 + 1. Let's put "1" bits into the corresponding boxes, and "0" bits in the rest:

Decimal:	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Binary:	0	0	1	1	0	1	0	1

Decimal 53 is equal to binary 00110101.

One final thing you should know is how to determine the largest decimal value you can store with a certain number of bits. If you have *n* bits, the formula is $2^n - 1$:

8 bits: $2^8 - 1 = 255$

16 bits: $2^{16} - 1 = 65{,}535$

32 bits: $2^{32} - 1 = 4{,}294{,}967{,}295$

At the time I write this, the personal computer industry is just about done phasing out 32-bit computers in favor of computers capable of slinging around 64 bits at a time. The highest decimal value you can store in 32 bits, as you can see above, is over 4 billion. The highest value you can store in 64 bits is… well, let's just say it's a big value.

Endianness

There is one further factor that complicates understanding binary, and it's called "endianness".

Any particular series of bits can be interpreted as two completely different numbers, depending on whether the series is meant to be interpreted as "little-endian" or "big-endian".

Say what?

As usual, it's best explained by example. Let's consider an arbitrary 4-bit binary number:

Bit #0	**Bit #1**	**Bit #2**	**Bit #3**
0	0	1	1

Based on everything we know so far, 0011 in binary is equivalent to the decimal number 3.

As we've seen, the decimal values of bit positions double as you add more of them. The bit representing the highest decimal value is called the "most significant bit", often abbreviated MSB. Predictably, the bit at the very opposite end is the "least significant bit", or LSB, because it has the least impact on the overall decimal value of the number, only capable of adding 1 to it.

So far I've presented every binary number to you in

"big-endian" format. Big-endian simply means that the most significant bit is the leftmost one. If bit #0 is the most significant bit in the number above, you end up with the decimal value 3.

Here's how the decimal values of each bit position break down in big-endian:

	Bit #0	**Bit #1**	**Bit #2**	**Bit #3**
Decimal value:	8	4	2	1

But what if we reversed the least and most significant bits of this number? What if we said, somewhat arbitrarily, that bit #3 was instead the *most* significant bit, and bit #0 the *least* significant bit? Well, then we'd have to flip around the decimal values of each bit so they went from lowest to highest instead:

	Bit #0	**Bit #1**	**Bit #2**	**Bit #3**
Decimal value:	1	2	4	8

So, what happens to our binary number 0011 in little-endian? Its decimal value magically changes from 3 to 12! The bit sequence itself didn't change, just the

direction we read it.

How in the world, then, are you supposed to tell if any given series of bits are big-endian or little-endian?

Most often this is determined by who makes the processor in the computer. Motorola processors, such as the 68000 or PowerPC series, have traditionally been big-endian by default. Intel processors and Intel-compatible processors have traditionally been little-endian by default. Just to make life more interesting, some processors can be re-configured to work either way. Thankfully for us, high-level programming languages shield us from this complexity most of the time.

It's merely the lowest level example of what computer scientist Andrew Tanenbaum was talking about when he said, "The nice thing about standards is that you have so many to choose from." You'll discover many more examples of this pervasive phenomenon in your programming career.

I, however, will try to keep things simple by using only big-endian binary for the remainder of this book.

Hexadecimal

With any luck, you're starting to see a pattern here. There's more than one way to represent any particular numeric value. It just depends on how many symbols are available to you.

We've covered good old decimal (base 10), and binary (base 2) which has much fewer symbols.

What about a numeric base with *more* than our beloved decimal's 10 symbols? One that comes up frequently in programming is hexadecimal, or base 16.

The first ten symbols used in hexadecimal (or "hex" for short) are exactly the same as those used in decimal:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

But then there are six more:

A, B, C, D, E, F

…for a grand total of 16 symbols.

Wait a minute, you cry! Those aren't numbers, they're letters!

The reality is that all 16 of them are just *symbols*. They are simply shapes to which we have assigned meanings. In this particular case, they represent the decimal values 0 through 15.

(For this book, I've used capital letters in hexadecimal

numbers. In the real world, they are sometimes lowercase. There's no real standard.)

Check this out: in hexadecimal, we can count all the way up to fifteen without having to add a second digit:

0

1

2

3

4

5

6

7

8

9

A

B

C

D

E

F

So, what's decimal sixteen going to look like in hexadecimal? You know the drill by now: increase the digit to the left by one, and wrap around back to 0:

10

The decimal number 16 is represented in hexadecimal as 10. Which also happens to be how the decimal number 2 looks in binary.

(A feeling of vertigo at this point is not unusual. It should pass.)

Let's press on with our hexadecimal counting. Next up is decimal 17:

11

12

13

14

15

16

17

18

19

We don't go to 20 yet, because we've still got symbols we haven't used!

1A

1B

1C

1D

1E

1F

Now we're out of symbols. So, increase the digit to the left, and wrap around to zero:

20

Hexadecimal 20 is equal to decimal 32.

So, what benefit is hexadecimal to you as a programmer, if binary is the computer's native tongue?

For one thing, hex is more compact than binary. I showed you how you can store a decimal value up to 255 in 8 binary digits. But, you can represent the same number in only *two* hex digits.

Binary 1111 1111 = decimal 255 = hex FF

It's also much easier for humans to convert between hex and binary than it is to convert either to decimal.

Any 4 bits can be represented in a single hex digit:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

As we know, a group of 8 bits is called a byte. So, any byte can be represented with two hex digits. Here's an example:

From the chart above, we can see that binary 1001 is hex 9, and binary 1100 is hex C. You can simply put these end-to-end:

1001 1100 = 9C

How easy was that?

The conversion goes both ways. For example:

B3 = 1011 0011

We could then put 1011 0011 into our imaginary "boxes" to convert the binary value to decimal:

Decimal:	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
Binary:	1	0	1	1	0	0	1	1

We have 1 bits in the following boxes: 128, 32, 16, 2, and 1. Simply add those numbers together:

128 + 32 + 16 + 2 + 1

= 179

So:

B3 (hex) = 1011 0011 (binary) = 179 (decimal)

All the same number, just written different ways.

You're now able to convert between three different numeric bases! Believe it or not, this is something programmers have to do all the time. Congratulations!

Signed numbers

There's an entire topic that I've been carefully avoiding. This whole time, we've only been talking about numbers greater than or equal to 0 – positive numbers.

Given that the only thing a computer understands is bits, and you've already seen how bits form positive numbers, what does a computer do when it needs to store or process a negative number?

It turns out it just looks at the same bits in a different way.

There are two types of numbers that every programmer must understand conceptually: "unsigned" and "signed".

You can think of these terms as referring literally to the presence or absence of a sign indicating whether the number is positive or negative.

If all you need to work with are the numbers starting at zero and going up, you can happily use an unsigned number. In an unsigned number, all available bits are used to describe the value of the number.

But if you need to represent a value that might be, at various times, either positive or negative, it must be a signed number. Signed numbers "steal" one bit to indicate the sign of the number — is it positive or negative?

That leaves us one less bit available to represent the

actual value, meaning signed numbers can only be half as large as an unsigned number with the same quantity of bits.

Let's look at a simplified example:

An 8-bit number, as we've already established, can be used to represent the values from 0 to 255. But only if it's interpreted as unsigned!

To represent a negative number, we steal one of those 8 bits to use as a positive/negative sign, leaving us with 7 bits. Using 7 bits, we can only represent 128 values, but in combination with the sign bit, we gain the advantage that the value can be either positive or negative.

To put it even more simply:

An unsigned 8-bit number can store values from 0 to 255.

A signed 8-bit number can store values from −128 to 127.

Note that regardless of whether the number is signed or unsigned, a full range of 256 values is achieved, but that range "moves". Unsigned 8-bit numbers count up 256 values from 0. Signed 8-bit numbers count up 256 values from −128.

This raises an interesting question. Say you have an unsigned 8-bit number, the maximum value of which is 255, and you try to add 1 more to it?

At that point the number has overflowed the space allotted for it. It would require more than 8 bits to represent decimal 256, but if all we have are 8, the best we can do is wrap around from 255 to 0.

If you were to write a program that tried to count from 0 to 300 using an unsigned 8-bit value, it would never finish running, because it would hit 255 and wrap around to 0 over and over and over again.

What about a *signed* 8-bit value that's already at its maximum value of 127 when you try to add 1? It wraps around to its lowest value too. Not 0, but −128. So our poor program trying to count to 300 is going to have an even worse time. (How would we solve the problem? By reserving more bits to store the counter in. A 16-bit number will get us well beyond 300, but it too has an upper limit.)

To understand why it is that, in signed 8-bit arithmetic, 127 + 1 = −128, we have to look at how the number is stored internally in the computer.

To represent the negative version of any particular number, the computer uses what is known as "two's complement". This is a very math-y way of saying something that's actually pretty simple.

Let's say we have 8 bits, into which we have stored the decimal number 5. In binary, it will look like this:

0000 0101

In two's complement, the most significant bit (in our case, the leftmost bit) is set to 0 for positive numbers, and 1 for negative numbers. So regardless of whether we read 0000 0101 as signed or unsigned, it's always decimal 5.

To change the sign of a number, we create its two's complement by following these steps:

1. Invert all the bits — convert 0s to 1s, and vice-versa.

2. Add 1 to the binary result.

Let's convert our 5 to −5.

We start with the bit pattern for 5:

0000 0101

Invert all the bits:

1111 1010

Add 1 to the binary result:

1111 1011

And that is the bit pattern the computer uses to store −5. But you and the computer must have an agreement in code to process that number as a signed number, because that exact same bit pattern has quite a different meaning if we read it as an unsigned number: it's decimal 251.

This is a frequent mistake made by even the most

experienced programmers. Let's say as part of your program, you had to count down from 5 to −5 by subtracting 1 repeatedly. If you have told the computer to use an 8-bit signed value for your counter, your program will work fine.

If you have by accident told it to use an 8-bit *unsigned* value, your program will never finish running. A negative value cannot be represented with an unsigned number. Here is what will happen to the counter's value if it is unsigned:

5

4

3

2

1

0

255

254

253

…and it will continue counting down until it gets to 0 again, at which point it will loop back around to 255, count down to 0 again, and continue like that forever. It can never reach −5, so the program will never end.

This is such a common mistake that many programming languages will try to warn you about it in advance, if they can determine that you are expecting an unsigned value to go negative. But it's impossible to catch every possible variation of this error, so it's important for programmers to always keep the issue of signedness in mind.

Observe that the same process of two's complement turns our −5 back into a positive 5:

1111 1011

Invert all the bits:

0000 0100

Add one:

0000 0101

And we're back to 5.

Hopefully now you've got a pretty good grasp on how two's complement works, but you may be asking yourself: why?

After all, if all it takes to make a number signed is to reserve one bit to be a positive/negative indicator, why go through all this extra rigmarole of inverting the bits and adding 1? Couldn't we just call the most significant bit our sign indicator, and treat the remaining 7 bits the same way we do for unsigned numbers?

Well, as you might expect, there's a reason it's done this way, and it's to make things easier for the computer — not you, unfortunately.

To understand why, take a look at the following sequence of numbers, and their representations in two's complement:

This is how a computer actually counts:

-2	1110
-1	1111
0	0000
1	0001
2	0010

What I want you to notice is that the binary numbers count up using the exact same "odometer" rule we've used throughout the book. It doesn't matter that the equivalent decimal numbers start out negative, hit zero, and then go positive. The computer just applies the same rule every time and the bit patterns magically correspond to the correct decimal values.

If we didn't use two's complement, and simply made the most significant bit indicate a negative number, we'd have this:

This is NOT how a computer counts:

-2	1010
-1	1001
0	0000
1	0001
2	0010

The computer would have to first look at the most significant bit, and then take one of two completely different courses of action based on whether that bit was 1 or 0, as well as a special one-of-a-kind case when the number hits zero.

For the computer to have to deal with that three-way decision every time it performed math on a number would add up to a considerable amount of time spent doing even basic calculations and much more complex circuitry in the computer's chips.

Two's complement allows the computer to process negative numbers pretty much the same way it processes positive numbers — quickly and efficiently.

Units for data storage

Previously, I explained the concept of "powers of two". Because of the nature of binary, you end up seeing these numbers all the time in computer-related materials. You start to recognize them almost subconsciously after a while: 256, 1024, 4096, and so on.

In the realm of computers, powers of *ten* also come up frequently. These would be the number ten multiplied by itself a certain number of times. (For example: $10^3 = 10 \times 10 \times 10 = 1{,}000$)

You have probably seen lots of computer jargon beginning with prefixes like kilo, mega, and giga. These are technical shorthand, used to represent large numbers.

Kilo, abbreviated with a lowercase k, refers to 1,000 (one thousand) of something.

Mega, abbreviated with a capital M, refers to 1,000,000 (one million) of something.

Giga, abbreviated with a capital G, refers to 1,000,000,000 (one billion) of something.

Note that these units can be expressed in terms of each other. A mega-something is a kilo of kilo-somethings. A giga-something is a mega of kilo-somethings, or a kilo of mega-somethings.

Where you'll see these prefixes most commonly is when programmers want to measure the size of a

particular piece of data, or the capacity of something that contains data, such as a hard drive, or memory.

However, programmers have a different view of these prefixes. A traditional scientist would use the 1k = 1,000 rule I've shown above. But since programmers work with computers, and computers have a natural tendency towards powers of two, we redefine 1k to mean 1,024.

And 1M to mean 1,024 × 1,024.

And 1G to mean 1,024 × 1,024 × 1,024.

Let's rewrite our chart for programmers, then:

Kilo, abbreviated with a lowercase k, refers to 1,024 (or 2^{10}) of something.

Mega, abbreviated with a capital M, refers 1,048,576 (or 2^{20}) of something.

Giga, abbreviated with a capital G, refers to 1,073,741,824 (or 2^{30}) of something.

How about some more examples? Let's start at the very bottom and work up. We already know that a bit is the smallest possible thing a computer can understand; a digital "atom" that can be either 0 or 1.

In modern computers, 8 bits is referred to as a byte. (Humorously, a 4-bit value is occasionally referred to as a nibble or nybble — after all, what else would you call half a byte? This is the sort of joke programmers make. I apologize on all of our behalves.)

Once you reach 1,024 bytes, we call that 1 kilobyte, or 1 kB. The lowercase k is our kilo abbreviation, and capital B refers to bytes.

At 1,024 kilobytes, we have reached 1 megabyte, or 1 MB.

We don't need to change units again until we reach 1,024 megabytes — 1 gigabyte, or 1 GB.

A couple decades ago, it seemed like science fiction to imagine we'd ever be talking about computer storage in terms of gigabytes. It was an unfathomable amount of space that would be virtually impossible to fill. As with most predictions of the future, we were wrong.

Hard drives now come in capacities of multiple terabytes (TB) which, as you might have guessed, are 1,024 GB each.

We're not quite there yet, but the next stop will be petabytes (PB) — or 1,024 TB. For those keeping score, that's a million gigabytes.

After that come the exabyte (EB), zettabyte (ZB), and yottabyte (YB), each representing 1,024 of the last. That should last us a while.

There's a weird side effect of the fact that engineers consider 1 kB to be 1,024 bytes and other types of people consider 1 kB to be 1,000 bytes. This tends to manifest itself when you buy, say, a 1 GB hard drive, and discover

that even without anything on it, it has less than 1 GB of free space available to you. What happened?

What happened was the people who wrote the marketing copy on the packaging told you that the hard drive had a 1 GB capacity, which was arguably true, because they meant 1,000,000,000 bytes. But that's not what a computer considers 1 GB. To a computer, 1 GB means 1,073,741,824 bytes.

That's a difference of over 73 million bytes (approximately 73.7 MB) of space that you inferred from the "1 GB" printed on the box you would be getting, but in fact are not. But, because of a technicality, you haven't *really* been lied to. Misled, possibly.

To try to resolve this classic confusion, a series of alternate prefixes were introduced: kibi, mebi, gibi, and tebi. Since kilo, mega, and giga have long been established in non-computer circles as meaning a thousand, a million, and a billion, respectively, the following new terms where introduced specifically for measuring powers of two:

1 kibibyte (KiB) = 1,024 bytes

1 mebibyte (MiB) = 1,024 KiB

1 gibibyte (GiB) = 1,024 MiB

1 tebibyte (TiB) = 1,024 GiB

These terms have been slow to catch on, simply

because the older, confusion-prone prefixes were already in broad use, and the newer terms sound, well, kind of weird to older ears. Even programmers, more often than not, still say “gigabyte” when they are really talking about a “gibibyte”. Bad habits are tough to break. A programmer will likely assume you are talking about a multiple of 1,024 if you use kilo, giga, or mega. When talking to non-programmers, you might need to be more specific.

This is all well and good for measuring data storage and capacity. There’s one other common variation though, and it’s typically used when measuring data transfer speed, such as how fast data goes through a computer network.

Units for data transfer

Earlier, I talked about how it's convenient for computers to speak binary, because 1 and 0 can easily be represented as the presence or absence of electricity on a wire. If you simplify the concept greatly, a "network" is nothing more than an electrical wire between one computer and another.

Computer #1 can put electricity on the wire to transmit a "1" to Computer #2, and turn off the electricity to transmit a "0". But for this to work, both computers have to have some understanding of how often these transitions occur on the wire.

If Computer #2 checks the wire and sees a "1", then it checks the wire again and sees "1" again, is it seeing a series of two "1"s, or is Computer #1 still transmitting the same "1" it was originally? There's no way to be sure.

But if through some prior arrangement, Computer #1 agrees to update the electricity on the wire 300 times per second, Computer #2 agrees to *check* the wire 300 times per second, and both start at the same time, then reliable communication can be performed. These two computers can be said to be communicating at 300 bits-per-second, or bps.

(Truth be told, that's a pretty gross oversimplification of serial communication, and if any professional network engineers are reading this, they're probably cringing right

now. More realistically, the actual bits of data are wrapped in even more bits designed to protect the integrity of the data bits from one computer to the other. Then, on top of that, there are additional wires that work like traffic signals to regulate the flow of data. But, at the end of the day, from a programmer's perspective, the bits just go over the wire and everyone's happy. Hopefully.)

When dealing with data transfer speeds, network engineers prefer to describe it in terms of bits, rather than bytes, due to what's physically happening on the network wire. Also, very early computers did not always agree on how many bits made up a byte. By talking strictly about bits, the issue became moot to the hardware engineers designing the network and was left for the programmers writing the communications code to deal with.

Note that a lowercase "b" is used to indicate bits, rather than the capital "B" we use to indicate bytes. Be very careful not to mix these up, as the difference can be quite significant.

In measures of data transfer speed, the unit prefixes are usually assumed to mean their powers-of-ten form. That is to say:

1 kilobit-per-second (Kbps) = 1,000 bits-per-second (bps)

1 megabit-per-second (Mbps) = 1,000 Kbps

1 gigabit-per-second (Gbps) = 1,000 Mbps

These units are used commonly to advertise, among other things, the speed of the connection you subscribe to from your internet service provider. They might advertise "6 Mbps down / 1 Mbps up", but what does that actually mean?

First of all, the "down" value refers to information coming "down" from the internet to your computer. "Up" refers to information going "up" to the internet *from* your computer. In that sense, you can think of the internet like a bookshelf just above your reach. To get something off the internet "bookshelf", you have to take it "down" and vice-versa.

Internet providers optimize their networks to provide greater downward speeds at the expense of upward speeds. It's typically much, much faster to download than to upload something on a consumer internet connection because, more often than not, downloading is what consumers want to do with their internet connection.

Our hypothetical provider offers 6 Mbps down. Sounds pretty good! But let's math it out.

First, we'll figure out what 6 Mbps is in bps:

6 megabits-per-second = 6 × 1,000 Kbps

= 6,000 Kbps

6,000 Kbps = 6,000 × 1,000 bps

= 6,000,000 bps

In our modern computers, there are 8 bits in a byte:

6,000,000 bps ÷ 8 bits-per-byte

= 750,000 bytes-per-second (Bps)

…and 1,024 bytes in a kilobyte:

750,000 Bps ÷ 1,024 bytes-per-kilobyte

= 732.42 kilobytes-per-second (kBps)

So, our 6 Mbps connection actually corresponds to about 732 kB (less than a megabyte) transferred per second. Suddenly, it doesn't sound quite so impressive. Maybe that's why they still use bits-per-second notation.

Floating point

Once again I've been hiding a particular topic from you, but you know enough now to be able to handle it!

Every number we've talked about so far has been an "integer". They've all been whole numbers like 1, 2, or 3. Never fractions like ½ or 2.78. Surely a computer has to be able to deal with fractions? Indeed they do.

At the risk of sounding like a broken record, I will remind you again that the only thing a computer understands is bits. So, what's needed is a way of representing a fractional number by using a particular bit pattern. One method of doing this is known as "floating point".

That's a funny name. What does it mean? It refers to the fact that the decimal point can "float" around in the number. It might float over to the left:

1.743485

Or over to the right:

174348.5

Or somewhere in-between:

1743.485

The most important thing you must know about floating point math is that, unlike integer math, it is imprecise.

Any computer will be able to add the integers 1 + 1 and always come up with 2.

But what about 0.152321 + 912.772? It's an easy enough problem to solve on paper, so it should be no big deal for a computer, right?

Unlike humans, computers are not able to calculate fractional numbers to an infinite precision. They can be programmed to have a *very high* degree of precision, but if you keep adding numbers after the decimal point, sooner or later, some rounding must occur, which will lead to a loss of accuracy.

From time to time, a newspaper article will pop up about a silly computer program that went awry because it, say, divided 1 by 2, and instead of getting 0.5, came up with 0.4999999999 and that in turn threw off some critical calculation somewhere else down the line and, long story short, the rocket went off into deep space instead of landing on the moon.

Why would that happen? I mean, the only thing computers *do* is math, right? How could they get such a simple computation wrong?

It all becomes a bit clearer when we look at how floating point numbers are represented in binary.

As you know, the more bits you can set aside for a number, the larger the range of the number you can store in those bits. That's true in floating point as well, but with

a twist: in floating point, more bits also mean greater precision.

A typical size used for floating point numbers is 32 bits. A 32-bit floating point number is referred to in most major programming languages as simply a “float”.

For situations where a much greater accuracy is required, the programmer may opt to use a 64-bit floating point number instead, usually called a “double”.

Let’s stick to 32-bit floats for now, since we’re just dipping into the subject. What you’ll learn applies similarly to doubles as well — they’re just able to hold larger, more accurate values than floats.

In math class, or perhaps on a digital calculator, you may have encountered “scientific notation”, which is a compact way of representing very large or very small numbers.

Using scientific notation, we could take a very large number, like 3,140,000,000 and write it instead as 3.14×10^{9}. Same number, just abbreviated.

Similarly, very small numbers, like 0.0000000031 could be written in scientific notation as 3.1×10^{-9}.

In these examples, the 9 and −9 are called the “exponent”, and the part before the multiplication sign is called the “significand”.

By the way, there’s a quick shortcut to multiply a

number (like 3.14) by a power of ten (like 10^9). You simply take the value of the exponent and move the decimal point to the right by that number of places. In the case of 3.14×10^9, the decimal point in 3.14 moves 9 places to the right:

3.140000000 Start

31.40000000 Moved right 1 place

314.0000000 Moved right 2 places

3,140.000000 Moved right 3 places

31,400.00000 Moved right 4 places

314,000.0000 Moved right 5 places

3,140,000.000 Moved right 6 places

31,400,000.00 Moved right 7 places

314,000,000.0 Moved right 8 places

3,140,000,000. Moved right 9 places

If the exponent is negative, just move the decimal point to the left instead.

Now that we've established what an exponent and a significand are, let's get back to floating point.

We've decided to use 32 bits in which to make our float. Let's take 1 bit and call it a sign bit to indicate whether the number is positive or negative. Let's take the next 8 bits for the exponent, and the remaining 23 for the

significand:

Sign	Exponent	Significand
1 bit	8 bits	23 bits

That accounts for all 32 bits.

This is the bird's eye view of floating point. There are a lot of really hairy details in the way floats are stored in binary, and I'm going to pretend they don't exist for the sake of this brief overview.

You can find these details on the internet if you really want to know, but virtually all programming languages shield you from having to worry about them. (A quick example is that there are special bit patterns that are reserved to mean "infinity" and "NaN". NaN means "Not a Number" and is the result of dividing any number by zero, among other things.)

Much more important than the bit-level details is your awareness of the issue of limited precision. Because floating point doesn't have infinite precision, the best it can do in many cases is store a very close approximation of your number. It's for this reason that programmers can't assume that, for example, 0.1 + 0.3 is exactly equal to 0.4. But it will be very close. The best thing to do in these cases is test against a very small range of numbers; for example we can reasonably expect that 0.1 + 0.3 will

be ≥ 0.399 and ≤ 0.401.

Suffice it to say, representing fractional numbers in binary is significantly more complicated than representing integers.

For one thing, forget about just using the "odometer" technique to count up to the next floating point number. Because the number has been split up into three separate chunks, it simply won't work.

Dealing with floating point is so much more complicated than dealing with integers that the processors in modern computers have dedicated circuitry (generically called an FPU, or floating point unit) just to handle them. Before FPUs came along, it took substantially longer for a computer to do floating point math than integer math, and programmers tried to avoid floating point if they could help it for speed reasons. This is not so much a problem today.

In the chapter on binary, I showed you how to convert binary numbers to decimal or hexadecimal. I'm going to skip that for floating point as it requires a level of detail that I feel is beyond the scope of this book while at the same time being less useful day-to-day than integer conversions. Converting floats between binary and decimal is honestly a task best left to the computer. Let your programming language take care of the details here, but do so aware of the inherent limitations of floating point and you'll be a step ahead of many programmers.

Fixed point

Let me twist your brain now once more. The processors in some computers do not support or understand floating point. This was true of some very early personal computers, and is still true of some of the small, inexpensive processors that you might find in commodity consumer electronics.

How then, using only whole, integer numbers, could you do any sort of computation that resulted in a fraction? Usually, the answer is: fixed point.

Just from the name alone, you can probably guess the major difference between floating point and fixed point. In fixed point arithmetic, the decimal point doesn't move around. You will establish a specific level of precision (say, 2 digits following the decimal point), and that will be that.

Probably the easiest way to explain fixed point arithmetic to humans is to use money as an example.

Storing monetary values in a computer program using floating point is a disaster waiting to happen. As we've seen, they're imprecise, and subject to rounding errors over time. There is no room for error when dealing with people's bank balances.

But money is fractional! At least, it is in many currencies, such as US or Canadian dollars, or British

pounds. If I have twelve dollars and fifty cents, you'd write that as $12.50. See? A fraction. How can we deal with amounts of money in a computer program without the accuracy problems inherent in floating point?

The answer is amazingly simple. We just think about the money in different units. Instead of "twelve dollars and fifty cents" we could say "twelve hundred and fifty cents". It's exactly the same amount of money. The only odd thing about it is that's not the way we're used to saying it. But it's the same:

$12.50 = 1,250 cents

Guess what? We got rid of the fraction. 1,250 is an integer number we can quite happily manipulate in a computer without any rounding errors.

If I deposit another 75 cents into my account, I just add: 1,250 + 75 = 1,325 cents. Or, more colloquially: $13.25.

This is exactly what fixed point is like. It's a scaling factor that you consciously decide as a programmer to use to avoid fractions.

In our money examples, we're scaling by 100. We multiply the dollar value by 100 to get cents, and from then on perform all our calculations in cents. If we're not dealing with money, it doesn't have to be 100. We can scale by any number.

Let's say we have a program that deals with weights in the imperial system — pounds and ounces. There are sixteen ounces to a pound, so we'll use a scaling factor of 16. Instead of saying something weighs 2 pounds, 3 ounces, we can (in the internals of our program) say instead that it weighs 35 ounces.

Now let's say our program is adding up the weights of two packages that both weigh 35 ounces, for a total of 70 ounces. But we don't want to tell the user the total is 70 ounces because that is not the way a human would expect to receive that information. A human would want to hear that the combined weight is 4 pounds, 6 ounces. But how do we get from "70 ounces" to "4 pounds, 6 ounces" without fractions?

We can use a combination of integer division and something called a modulo.

If we were to precisely divide 70 ounces by 16 to get the weight in pounds, we'd get 4.375 pounds. That's accurate, but again, few humans would understand what .375 of a pound is. Most would be unable to do the math quickly in their head to convert .375 pounds to 6 ounces. Our program needs to do better.

The first step is to use integer division instead of the traditional division we learned in school. The sometimes-handy thing about integer division is it completely discards the fractional portion of the result. In integer division, 70 divided by 16 is not 4.375, but simply 4.

That is to say, the maximum number of times that 16 can go into 70 without resorting to a fraction is 4.

You may remember from doing long division in elementary school that when a number can't be divided evenly by another number, you end up with a leftover piece, called the remainder. We need to calculate the remainder of 70 divided by 16.

We've already determined, using integer division, that 16 can go into 70 fully 4 times.

16 × 4 = 64

And how much is left over if we subtract 64 from 70?

70 − 64 = 6

That's our remainder, 6.

Thus, 70 ounces = 4 pounds, 6 ounces.

We can save ourselves a bit of time when doing this calculation in the future by learning about the modulo operation.

Modulo is a feature offered by most major programming languages, and its purpose is to compute the remainder of an integer division in one easy step. It is often represented by a percent sign (%) in code. Going back to our weight example, we wanted to convert 70 ounces into pounds and ounces, given that there are 16 ounces in a pound:

70 ÷ 16 = 4 (using integer division)

70 % 16 = 6

Boom, just like that. 4 pounds, 6 ounces. And at no point did fractions have to come into play.

There's another way to handle fixed point, and that's to simply reserve a set number of bits in your number to represent the fractional portion.

Let's say I have an 8-bit unsigned number. It should be old hat by now that this can store any number from 0 to 255.

But there are other ways I could divvy those bits up. What if I reserved the least significant bit for a fraction? That would mean I could effectively represent numerical increments of 0.5, at the cost of reducing my largest possible number to 127.5. How? Watch:

0000000 0 = 0.0

0000000 1 = 0.5

0000001 0 = 1.0

0000001 1 = 1.5

0000010 0 = 2.0

…

1111111 1 = 127.5

etc.

All I'm doing is treating the first 7 bits as a regular old unsigned binary number, and using the least significant bit to indicate that I should add an additional value of "0.5" if it's set to 1.

What if that's not precise enough? What if I wanted to increment by 0.25, so there would be 3 intermediate steps between each whole number instead of just 1? Well, if I can live with a maximum value of 63.75, I can simply reserve another bit off the tail end to put towards the fractional piece of my number:

000000 00 = 0.0

000000 01 = 0.25

000000 10 = 0.5

000000 11 = 0.75

000001 00 = 1.0

…

111111 11 = 63.75

And so on. You can fake varying degrees of precision just by reading the same bit patterns differently. You could have a 16-bit number with 4 bits dedicated to the fractional part. Or a 32-bit number with 8 bits dedicated to the fraction. Or whatever you like. The only questions you'll need to ask yourself are:

1. What is the largest number you'll need to represent, and do you have enough bits to hold it?

2. How precise do your fractions need to be, and are there enough bits left over to achieve that precision?

Conclusion

You've learned an incredible amount about computers and how they count over these pages. Give yourself a pat on the back! It's OK if you don't fully understand everything here on the first reading. It will make more and more sense as you continue to learn about programming, and you can always refer back here for a refresher.

It may be hard for you to understand at this point what all this math has to do with what you no doubt want to accomplish: putting windows on the screen, drawing graphics, playing sounds, and responding to user input. At this point the best I can offer is: trust me, it's all relevant. Now that you know the bottom layer of our programming cake, we have a common language I can speak to you in for the next volume in this series.

I hope you have enjoyed re-learning how to count, and I look forward to meeting you again in book two!