# What is programming?

(somewhat) Gentle introduction for complete laymen into computer world

Khambar Dussaliyev

March 5, 2023

This essay is intended to answer (very briefly) to a question: *What is programming*? Although computer programs nowadays are pretty ubiquitous in nature, they remain being *black boxes of magic* for most people, even tech-savvy ones.

However, being a *black box of magic* to some extent is a requirement for some commercial software (especially when there are trade secrets involved) natural. But this essay is not intended to answer how every software works in details, but *how every software works, in general*.

**DISCLAIMER: This is a preliminary DRAFT. If you've found some errors, or somehow interested in this work, please write to anuarkaliyev23@gmail.com**

# Contents

# Chapter 1

# Introduction

Basically, *everything* you do on a computer is running some program. Everything, no matter what you do is a result of executing and working with some program. Your OS[1] *is a program*. Your internet browser is a program. Your media player, file explorer, video game, media editing software, office software *are programs*. *Everything that allows you to interact*[2] *with computer* (*and not just bare metal*) *is a program*.

So, essentially, *what is a program*?. According to Merriam-Webster[3] definition[4] (applicable for us), we can use following as a definition:

> program is a sequence of coded instructions that can be inserted into a mechanism (such as a computer)

The gist of it being — *sequense of instructions*. So every interaction we can possibly have with a computer is somehow just a set of instructions. But *how computers do understand out instructions*? If I just shout into the microphone some command, computer will not just do as I say[5]. The same effect will have some instruction that I carefully write them in some text document, using Microsoft Word, for example.

How do I make computer to understand what I want from it? To understand this, *we must first understand what is a computer*.

[1] Operation System — Windows, Linux-based, MacOS, Android, iOS etc.

[2] Interaction with a computer here and on will not take physical interaction with a bare metal in account

[3] Here and on Merriam-Webster dictionary is referred to as a 'general-scope' dictionary to avoid technical details redundant for this essay

[4] https://www.merriam-webster.com/dictionary/program

[5] Provided, there is no running program, responsible for such behavior

# Chapter 2

# What is Computer?

## 2.1 It's all about information

Let's once again refer to Merriam-Webster dictionary for a *computer* definition [1]:

> computer is a programmable usually electronic device that can store, retrieve, and process data.

So, a computer directly tied to all sorts of *data maniplation*. But what is data, and how can it be manipulated? Data is pretty much any factual information. Weather outside a window? Data. T-Shirt you're wearing? Data. Dusty books in my shelves? Data. But there are a certain layers present. The fact that I'm wearing a T-Shirt is data (even if I'm not — also data). What color it is is also, most certainly data. Is there any text or image present? Both the existence and *information* in it — also data. Let's also not forget about colors, sizes, fonts. We are living and have always lived in an enormous ocean of data. Nowadays, with our technology even more so.

But do we have use for this data? Well, the answer is — it depends. To assess data without well-defined goal will almost always result in you drowning in said data without much progress, since world offers us practically indefinite source of it. We must put our data into some perspective, some context. Once we put our data in some context and it becomes *useful* for us, *it becomes information*.

> To somehow navigate in this world, we must put our data in context. Data with given context becomes somewhat useful. Such data called *information*.

Information is much more well defined than data. *We can measure information, actually.* We even have a science discipline, called *Information theory*, that researches all about information, from mathematical and engineering

[1] https://www.merriam-webster.com/dictionary/computer

standpoint. Not only that, but we have an entire industry, called *Information Technologies* built entirely on a foundation provided by Information theory and adjacent disciplines.

But until we dwell into technical stuff, we must also remember, that *we used to operate with information*. Our brain is a natural computer, disecting data into information that we use in our everyday life. How come I am so sure to call that information and not just raw data? Well, that greatly depends on a scale, but *our brain have very defined goals*. One of the main said goals being to *keep us alive*. Therefore there is a context, and brain will always tend to categorize things (organize data) based on this goal[2], making it somewhat useful, therefore making it an *information*. Even the simple fact, that we don't notice our noses, although our eyes do see it constantly, tells us how natural brain is in working with information.

So, being a natural organic computer, we must first understand what we do with information, to have insight on what computers do with information. All processing of information, our minds in work is mostly an internal process. *Sooner or later there is a point, when we must exchange said information*. So, how do we exhange it? We have an enormous number of ways, actually. We can say something to another person, write it down, pass via somebody a note, we can just hint at something. We can send message in a messenger, send an email, send a radio-signal, we can knock a Morse code. *We are practically limitless with one major nuance — the other side must be able to understand us*. There is no point in sending email to a person, who can't use it[3].

We now can conclude one fundamental distinction: information itself is mostly independent from it's carrier. To demonstrate it more clearly, let us consider following example: I want to send information to my friend at the table, with the main message being '*pass me salt*'. There is definetely a context: we are at the table, eating food, there are at least two parties involved (me and my friend), and I expect salt to exist somewhere at the table. I can pass this information with a various number of ways, provided my friend understands me. Just to mention a few:

- Say to him 'pass me salt'
- Ask him to pass me salt in other language, he is familiar with
- Write a note to him 'pass me salt'
- Write a note in foreign language, he is familiar with
- Get his attention and non-verbally point at salt
- Write him a message in messenger, expecting phone to be near them

[2]If your goal at the moment being something more specific, than just stay alive, many of the information brain gives you still raw data

[3]In general. Sometimes we are interested only in sending information, not concerned by an actual delivery. Some legal procedures can be of an example

- Get his attention and use ASL or alternative, provided he is familiar with it
- Exclaim obviously 'Oh! This food will be so much better with salt! I wish somebody passed it to me now', provided he understood our hint
- Rhytmically knock with Morse code, provided he understands it

In all aforementioned examples we can clearly see, that the gist of our 'message' stayed the same. *We did pass a more or less the same information* in each and every case. Despite the medium being completely different, if our friend can undestand us, nothing really changed for him or us. In such cases the 'main message' containing an actual useful information we are willing to exchange usually colloquially called a '*payload*'. However, despite our 'payload' being virtually the same, we did pass some additional information (or data — depending on context) along the way, didn't we?

Let's use an above list one more time, but will provide additional few details, just for example:

- Say to him 'pass me salt'
  - In what voice tone?
  - How loud did we ask?
  - What face expressions followed along our request?
  - Was there any gesticulation involved? How intense?
  - In what speed we asked?
- Ask him to pass me salt in other language, he is familiar with
  - What language?
  - Was there any context in using this language?
  - In what voice tone?
  - How loud did we ask?
  - What face expressions followed along our request?
  - Was there any gesticulation involved? How intense?
  - In what speed we asked?
- Write a note to him 'pass me salt'
  - What font did we use?
  - Is it hand-written?
  - Font color?
  - Font size?
  - Paper type?

  - Paper size?
  - Was paper plain white, or was it with pictures?

- Write a note in foreign language, he is familiar with
  - What language?
  - Was there any context in using this language?
  - What font did we use?
  - Is it hand-written?
  - Font color?
  - Font size?
  - Paper type?
  - Paper size?
  - Was paper plain white, or was it with pictures?

- Get his attention and non-verbally point at salt
  - Were we mumbling at the same time?
  - How we got his attention? Tapped his shoulder? How strongly?
  - How did we point? With a finger, palm, node?
  - What face expressions have we used?
  - How fast did we do it?

- Write him a message in messenger, expecting phone to be near them
  - Have he seen us typing a message?
  - How fast he reacted?
  - Was there any notification
  - Did we send an emoji?
  - Did we send some attachment?
  - We sent one message or several?
  - Did he read it?

- Get his attention and use ASL or alternative, provided he is familiar with it
  - How exactly did we phrase our request? By letters or by gests?
  - How fast did we transmit?
  - We followed along with our lips?

- Exclaim obviously 'Oh! This food will be so much better with salt! I wish somebody passed it to me now', provided he understood our hint

  - What intonation did we use?
  - How loud did we say it?
  - Where to did we look?
  - Do we have any specific accents?
  - Was there an emphasis on some words?

- Rhytmically knock with Morse code, provided he understands it
  - What period of time we used as an interval?
  - Did we repeat our message? How many times?
  - On what surface did we transmit?
  - Did we use our knuckle? Spoon? Knife?

So, we can conclude, that despite our *payload* being practically the same, we did pass additional information along with it. To put it into perspective, It's somewhat similar, as if we were asked to describe an envelope, it's size, stamps on it and additional notes, ignoring the payload, being a letter inside said envelope. Such auxiliary information, which is more often than not isn't of our interest, however *can be useful in certain scenarios*. Such data usually describe optional information aboout the *payload* itself, or details of how it was delivered. Such data usually called *metadata*

> The information itself, that we wish to store or exchange colloquially called a *payload*. Some additional details, that might be useful, regarding that information, but not tied to it directly usually called *metadata*

Let's say, I am writing a document in Microsoft Word. The *payload* here being anything, I typed directly in this document. However, once I saved it, not only the document itself was saved, but also a bunch of additional *metadata*. It can include date of the document creation, author[4] of the document, last save date, last print date, etc.*The same logic is applicable for virtually any file you've ever created*[5].

[4]Usually currently active user on OS level

[5]Sometimes, ability to control metadata becomes crucial to save sensitive and personal information. Some professions can put you in physical danger, if you're not cautious enough

## 2.2 You can't manage what you can't measure

Once we have gotten ourselves these neat definitions about data and information, we will not benefit from them until we resolve one fundamental issue: *How do we measure information*? How can we possibly find an adequate solution to count something so abstract in nature.

Well, first of all this is where this fine distinction between *data* and *information* comes in play. You see, *defining context* to transform our data into information provided us with one *fundamental advantage*. To demonstrate this, let's consider an example: I have a drawer, where a bunch of my T-Shirts is stored. Let's say, there is no order whatsoever and once I open the drawer, any of my shirts can be on top[6]. Let's say that I for a fact know that there is no more than 12 T-Shirts of different colors totally in my drawer. I open a drawer and see a green T-Shirt on top. *How much information did I receive*?

To answer this question, let's investigate how any data is measured, for starters. Well, universally in information theory there is a number of data measurement nomenclatures. The most ubiquitous and universal one being *bits*. It got it's name from a 'binary digit'. The binary part represents a tiny set of values it can be equal to, consisting only of {0, 1}.

Bit is a unit of information. It can be equal to either '1' or '0'.

Bits are the most basic and fundamental part of any computer-related information operation. *Everything* on your computer is stored in bits. Yes, *everything*. Everything you type in your Word documents, PowerPoint presentations, every audio track you've ever played, every site you've ever visited, every file you've created, *everything*. This simple fact might leave us with a bit of confusion: *How on Earth can we measure apparently everything with such basic unit, only accepting '0' and '1' as a value*?. The answer is – well, pretty easily. Once we have something, that cannot possibly be presented as value in a set of {0, 1}, *we just increase the number of bits to store that additional information*.

Consider 1 bit. Total number of values it can represent is limited to 2, by definition. It's either '1' or '0'. Well, if we increase the total numbers of bits and consider we have 2 bits. Now we have 2 possible 'cells', each of which can 'hold' 2 possible variants. Those values being: '00', '01', '10', '11'. Now, if something is still bigger than one of four possible variants, we can add bits until it will finally be enough. With a little bit of discrete math we can say for sure, that total possible variants a sequence of bits can hold equals to $2^x$, where 2 is a number of possible options a bit can 'hold' ('0', '1'), and $x$ is the number of bits we are ready to *allocate*.

To count something in bits is pretty similar as to how we would count something normally, but with one nuance. In a modern world, humans mostly calculating everything in a $base_{10}$. This simply means, that we have *10 digits* that construct *all of our numbers*[7]. Those digits being: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

[6] I would call that scenario uncanningly realistic

[7] There are examples of cultures, that used different number as their base. However $base_{10}$ is the ubiquitous one nowadays

| Number of bits | Maximum options | Options |
|---|---|---|
| 1 | 2 | {0, 1} |
| 2 | 4 | {00, 01, 10, 11} |
| 3 | 8 | {000, 001, 010, 011, 100, 101, 110, 111} |
| ... | ... | ... |
| $x$ | $2^x$ | { $x$ times 0, $\cdots$, $x$ times 1} |

Table 2.1: Total possible permutations of bits

So, our *digits* can represent one of at-most 10 options. But how can we describe something more than 9? Well$\cdots$ just start over and add additional digit! This is the same principle we follow, when we are counting in a $base_2$, working with bits. We can use pretty much any base we want, but along with $base_2$, $base_8$ and $base_{16}$ can be seen used widely[8]. So we can pretty much convert number in any base to any other base, always keeping consistency:

| $base_{10}$ | $base_2$ | $base_8$ | $base_{16}$ |
|---|---|---|---|
| $0_{10}$ | $0_2$ | $0_8$ | $0_{16}$ |
| $1_{10}$ | $1_2$ | $1_8$ | $1_{16}$ |
| $2_{10}$ | $10_2$ | $2_8$ | $2_{16}$ |
| $3_{10}$ | $11_2$ | $3_8$ | $3_{16}$ |
| $4_{10}$ | $100_2$ | $4_8$ | $4_{16}$ |
| $5_{10}$ | $101_2$ | $5_8$ | $5_{16}$ |
| $6_{10}$ | $110_2$ | $6_8$ | $6_{16}$ |
| $7_{10}$ | $111_2$ | $7_8$ | $7_{16}$ |
| $8_{10}$ | $1000_2$ | $10_8$ | $8_{16}$ |
| $9_{10}$ | $1001_2$ | $11_8$ | $9_{16}$ |
| $10_{10}$ | $1010_2$ | $12_8$ | $A_{16}$ |
| $11_{10}$ | $1011_2$ | $13_8$ | $B_{16}$ |
| $12_{10}$ | $1100_2$ | $14_8$ | $C_{16}$ |
| $13_{10}$ | $1101_2$ | $15_8$ | $D_{16}$ |
| $14_{10}$ | $1110_2$ | $16_8$ | $E_{16}$ |
| $15_{10}$ | $1111_2$ | $17_8$ | $F_{16}$ |
| $16_{10}$ | $10000_2$ | $20_8$ | $10_{16}$ |
| ... | ... | ... | ... |

Table 2.2: Numbers in different bases

[8]Writing everything in $base_2$ can be slightly inconvenient, when exchanging information with other programmers. Those bases gives us ability to 'shorten' binary code. Since bases are $2^3$ and $2^4$ respectively, we can 'group' together bits in a group of 3 (in the case of $base_8$) or 4 (in the case of $base_{16}$), making it easier for a human reading. Computer still operates with them in 'raw' $base_2$ format

This concept was wonderfully explained in Charles Petzold's book *Code*: *The Hidden Language of Computer Hardware and Software*. He explains different number bases with a following concept: we, humans, use $base_{10}$ mostly because it's very conveniently translates to *number of our fingers and toes*. So, a $base_2$ would be only natural, if math was invented by dolphins! Since they have only two flippers from each side it would be super convenient for them to calculate.

To convert $base_{10}$ number to $base_2$ number, you can consider following method:

1. Divide your number by 2. You need to store both the *quotient* end *remainder*.
2. Take your quotient as a number and repeat steps 1–2, until you got zero as a quotient
3. Write down all your remainder in *reverse order*. This is your binary number.

e.g.
We want to transform $35_{10}$ into binary form:

| Step # | Number | Quotient | Remainder |
|---|---|---|---|
| 1 | 35 | 17 | 1 |
| 2 | 17 | 8 | 1 |
| 3 | 8 | 4 | 0 |
| 4 | 4 | 2 | 0 |
| 5 | 2 | 1 | 0 |
| 6 | 1 | 0 | 1 |

Table 2.3: Step-by-step operations to convert $35_{10}$ to $base_2$

Now we got a list of *remainders*: {1, 1, 0, 0, 0, 1}. Once we put them in *reverse* order, we got our binary form: $35_{10} = 100011_2$

One might wonder: But why choose $base_2$ number system for computers in the first place? Well, the answer is, like many things, a result of overcoming real-world limitations. Any computer utilises some hardware, tangible, physical metal parts. Since computers nowadays are mostly electronic devices it made sense to create system detecting presence or absence of some electric signal. Presence of said signal would be connoted as '1' and absence as '0'.

Now, since we dwelled enough into subject, it's time to answer to the initial question of the subsection: *How much data did I receive, seeing a green T-Shirt on the top, once I opened my drawer*. We can even add some precision to this question, asking not *How much data* ⋯, but *How many bits of information* ⋯. Well, the fundamental *advantage* of data being in context, that I claimed in the beginning of the section is that now we know *total number of options*.

Since we know, that I only have 12 T-Shirts (and we don't really care about any other clothing in the drawer), there 12 possible options this could've ended. To conclude the total number of bits this information will occupy, I can use something called *Hartley function* to find out. It can be represented[9] as: $2^x \geq N$ with $N$ — representing our total outcomes number, 2 — representing possible options for one bit and $x$ — *number of bits we need to store it.* Mind '$\geq$' here — since bit is indivisible measurement, we cannot just take 1.5 bits, for example. *All cases ending up with a non-integer number need to be rounded up.*

Since we know our total number of possible outcomes — 12, we can calculate total numbers of bits we need to store that information, being the nearest power of 2 exceeding possible outcomes number.

> $2^4 \geq 12 \implies$ we need *4 bits* to store information about *any* T-Shirt being on top.

Please, mind an important detail: we need 4 bits *regardless* of which exactly T-Shirt will end up on top. To represent which T-Shirt it was exactly in each case of me, opening my drawer (provided, that total number of T-Shirts is constant), we have to come up with a *code*.

> Code — a system of signals or symbols for communication[10].

We must assign to every one of my T-Shirts a special *system of symbols*. Since information is measured in bits, it's only natural for our 'symbols' to be simple bits sequences. Number of bits in these sequences will be equal to 4, since $2^4 \geq 12$
Let's come up with a system of codes for T-Shirts:

[9] This form is somewhat simplified for the sake of clarity. Full Hartley Formula can be denoted as: $H_0(A) := \log_b |A|$

[10] https://www.merriam-webster.com/dictionary/code

| T-Shirt | Code |
|---|---|
| Red | 0000 |
| Brown | 0001 |
| Green | 0010 |
| Yellow | 0011 |
| Pink | 0100 |
| Gray | 0101 |
| Purple | 0110 |
| Blue | 0111 |
| Black | 1000 |
| White | 1001 |
| Orange | 1010 |
| Beige | 1011 |

Table 2.4: Our *encoding* for T-Shirts

So, upon opening my drawer and seeing my green T-Shirt on top I received 4 bits of information. If my drawer (or the T-Shirt itself) could send me binary information, they would send me '0010' representing it's color in this particular context.

Thus, we created an *encoding* for our T-Shirts.

## 2.3 Simplification through Standartisation

### 2.3.1 Scalability

So, we now know, how to use *Hartley function* to speculate about information's size. This is not, by any means, the only one method we can use to calculate information's size. However, It's pretty simple and kinda feels 'natural' to use. *Hartley function* gives us pretty good results when we are talking about somewhat randomly distributed options. Like in a case with a drawer and T-Shirts, I stated, that T-Shirt on top is random in nature. There is no hidden mechanism or consistent pattern we can observe, that can 'hint' us about which T-Shirt will be on top[11]. There are more applicable information measurements for the cases, when options are not evenly distributed. Also, there are numerous ways to 'shrink' space allocated for data. Researches focused on data compression algorithms are very important and it's hard to imagine some segment of IT, where this topic wouldn't be fundamental. But nonetheless, one way or another, understanding even something as simple as Hartley function and bit structures gives us major insight into how computers work and 'think'.

But do we really need to calculate how many bits of data will it occupy to do something and create own encodings for everything all the time? Well··· — it depends. It's a rare occasion to create our own encodings, as we did in the T-Shirts example. Imagine a world, where everybody creates his own encoding for all sorts of things only to solve a very narrow set of problems. Imagine everybody, from all over the world, to create own encodings for something as basic as date and time[12], text, all sorts of media, satellite signal processing, network communication and all sorts of other things — it would be a nightmare! Not only it's unreasonably hard and requires an immense quantity of man-years, but all that everybody would come up with will always be kind of bad quality. Encodings created to differentiate colors of T-Shirts won't be adequate to differentiate colors of pixels on the screen. Encodings, created for date and time will suffer lacking of different calendars and timezones, if at all support any. We couldn't pass a video to friend without passing special program developed for this particular video along with it. It will be simply unsustainable.

This caveat, regardless of the activity type, is solved by pretty much the only practical way possible — standartisation. Standartisation is *absolutely necessary* when you want something to be scalable at all. For example, in the early $19^{th}$ there wasn't standartised time in North America. It was simply not necessary, every little town had it's own clocks, that were adequate for people's needs. However, it would make some scheduling *between two towns* almost impossible. However, there wasn't much activities that would involve a simultaneous action between two towns. Travelling between them would take days, so why would people even bother with checking which hour it is in another town? It all have changed, once the trains and railroads came into place. *Now*

[11] Example of such pattern could be scenario in which I always see red T-Shirt on top, if on the previous opening I've seen on top the yellow one.

[12] Like we haven't enough problems with them already!

*it was actually important*. Transport companies had to organize trains somehow and create predictable schedules for all passengers and cargo to be transported and it *required* some standartised measure of time. Now we can have some video-conference meeting from several points all across the globe without any issues, since the time itself is standartised. All different calendars have predictable and standartised conversion operations and it's no problem to know what's time is it anywhere on the globe, in any standard calendar you like. We even have time standards for marking extraterrestrial observations. Our governments almost always impose some regulations on food, medical supplies, electrical devices, architecture and all sorts of any other things we can consume, produce and sell in our countries.

Most of the products and services in IT segment are way more scalable than fields requiring physical manipulation and processes. You can download a range of commercial and non-commercial products all over the world, provided you have a computer with stable internet connection. Once some method of data compression was discovered in North America, for example, there is not much stopping for any company in the world to use it[13]. Once new technology emerged, almost anywhere, anyone with adequate skills and knowledge can use it. It is the result of scalability, provided by standartisation. Not only that, but this environment creates a positive feedback loop, encouraging us to improve and evolve our standards, giving us even more scalability.

It wasn't always like this, though. Not so long ago, computers weren't as widespread as they are today. Back in a day, only some prestigious universities could have *one* giant computer for the whole university to use. By nowadays standards, they couldn't be called even remotely of good performance. IPhone 12 Max Pro, for instance has up to *16 billion times* more disk space, *1.5 million times* more RAM, while being almost *159 times* lighter than *Appolo 11*[14] *Guidance Computer*.

### 2.3.2 Nomenclature and text encodings

So, back in a day, when computers were ineffecient, programs *had to* be effecient to somehow compensate for hardware limitations. For this reasons, then it made more sense to 'invent a wheel' of sorts due to ineffeciency. It was easier to create some 'custom-built' encoding, that works well for some particular type of computer (sometimes even one specific computer), than to effeciently use some universal encoding of sorts. It was simply not feasible to create, for example, a text encoding where virtually all languages would be supported. But nonetheless, there was some standartisation, that eventually became widespread enough to become a basis for everything else.

Text encodings are one of the most fundamental standards there is in programming world. One of the first *text encodings* that became popular and *still widely used today* is ASCII[15]. Originally, it used 7 bits of information per symbol,

[13] In this paragraph, we suppose, that there is no special hardware or legal obligations involved
[14] This program succesfully landed on the Moon
[15] IANA prefers to call it 'US-ASCII'

producing $2^7 = 128$ possible variants for encoding characters. This provides us with 26 uppercase[16] letters, 26 lowercase[17] letters, 10 digits, some punctuation and additional symbols[18], and a bunch of control characters, which not always make sense to human, but are handy for the computers[19]. Since computers are built almost entirely on bits, it encourages engineers make things $2^x$ based, to use provided space more effeciently. So, after some time, ASCII began to require 8 bits[20] to encode single character. It was one of the major reason for currently used size nomenclature to emerge[21]. Sequence of 8 bits started being adressed as a *byte*. There were some computers, that weren't following '8 bit = 1 byte' structure, but 8-bit based computers became more popular, eventually. Although there were some historical 'debate' as how many bits a byte should consist of, nowadays

$$1 \text{ Byte} = 8 \text{ bits}$$

Eventually a nomenclature was formed:

| Name | Denotion | Size |
|---|---|---|
| Byte | B | 8 Bits |
| Kilobyte | KB | $2^{10}$B |
| Megabye | MB | $2^{20}$B = 1024 KB |
| Gigabyte | GB | $2^{30}$B = 1024 MB |
| Terabyte | TB | $2^{40}$B = 1024 GB |
| Petabyte | PB | $2^{50}$B = 1024 TB |
| Exabyte | EB | $2^{60}$B = 1024 TB |
| Zettabyte | ZB | $2^{70}$B = 1024 EB |
| Yottabyte | YB | $2^{80}$B = 1024 ZB |

Table 2.5: Nomenclature of bit capacity

### 2.3.3 Perception differences

Since we know, that space being occupied on computers directly tied to a total number of variants, we can now embrace some weird things, we can encounter in programming. For instance, consider following examples

- $9 + 13 = 22$

[16]Capital letters
[17]Small letters
[18]such as '%', '&', '+', '-', '!', different types of quotes and parenthesis
[19]For example: whitespaces, backspaces, tabulations, new line sequences, page breaks, etc. They can also represent some communication metadata.
[20]Since $8 = 2^3$
[21]https://stackoverflow.com/questions/42842662/why-is-1-byte-equal-to-8-bits

- Cats are believed to have 9 lives
- It just so happens, I have an extra 9.5$
- $9.8 + 0.2 = 10$

In each of these cases, we can encounter '9' in some form. If we consider all previous examples to be an ASCII text, we can roughly estimate data it would take up in a computer. We just need to count all symbols (including whitespaces!) and multiply it by 8. We know the encoding, we know a discrete number of variants we can come up with for a single characters. *But what if it is not a text*?. What if we need '9' *as a number, not text symbol*? There is no finite total count of all the numbers, they are infinite! And even more so, they are infinite both in ascending and descending orders! What should we do?

Examples like these show this difference, between a human mind and a way of how computers work. See, we are somewhat capable to construct our thesises and thoughts based on some *abstractions*. We made up a bunch of abstractions to make our life easier in some way or another. We can operate on abstractions, we are not required to have a complete and unambigious definitions on everything[22]. We can interpret '9' in various different ways not having to comprehend an enormous work of our brain behind it. For better, or worse, *computers work in fundamentally different way*.

For computers to compute (pun intended) we must firstly to *make up a finite set of possible variants*. This is usually done by creating several different 'types' of numbers[23]. Most of the times we just allocating different number of bytes to numbers, inherently making these numbers elements of the finite set. Generic example of this can be seen as:

| Name of the type | Size | Possible values |
|---|---|---|
| Pretty-Small-Number | 1 Byte | $x \in [0..2^8 - 1]$ |
| Not-So-Small-Number | 2 Bytes | $x \in [0..2^{16} - 1]$ |
| Generic-Number | 4 bytes | $x \in [0..2^{32} - 1]$ |
| Pretty-Big-Number | 8 bytes | $x \in [0..2^{64} - 1]$ |
| Very-Very-Big-Number | 16 bytes | $x \in [0..2^{128} - 1]$ |

Table 2.6: Example number type sizes

But wait — we cannot create negative numbers in this example! Should we use structure just like this for negative numbers also? We can, but it's not really convenient. However, we can reserve one our bits for the sign! Thus, we will not change total number of variants these bits provide us, but will change encoding for each number, subsequently 'shifting' our set of possible values:

[22]We do tend to unambiguity in many cases, though. Especially in science.

[23]Programming Language types is a vast and fundamental topic, and not limited just to numbers.

| Name of the type | Size | Possible values |
|---|---|---|
| Pretty-Small-Number-With-Sign | 1 Byte | $x \in [-2^7..2^7 - 1]$ |
| Not-So-Small-Number-With-Sign | 2 Bytes | $x \in [-2^{15}..2^{15} - 1]$ |
| Generic-Number-With-Sign | 4 bytes | $x \in [-2^{31}..2^{31} - 1]$ |
| Pretty-Big-Number-With-Sign | 8 bytes | $x \in [-2^{63}..2^{63} - 1]$ |
| Very-Very-Big-Number-With-Sign | 16 bytes | $x \in [-2^{127}..2^{127} - 1]$ |

Table 2.7: Number types, supporting negative numbers

It's all good and all, but these examples show only *integer* numbers. What if we have some number like 9.23? We cannot store it in any type of the aforementioned tables — none of them support fractions.

We can make up a type, that works with *fractions*[24]. For example, let's create a type taking 2 bytes. In this case we can define, that $1^{\text{st}}$ byte will represent an integer part and a $2^{\text{nd}}$ byte will represent a fractional part. We can use an interesting property of numbers to achieve our goal: You see, we can present any number as a sum of digits, multiplied by it's *base* in the power of digit's ordinal number.

Let's take a number, for example — 1024.

> $1024 = 1 * 10^3 + 0 * 10^2 + 2 * 10^1 + 4 * 10^0$. Mind that, this method works not only for $base_{10}$ numbers, but for any $base_x$, you should only correct multiplier for digits. This will also convert number from $base_x$ to $base_{10}$ in the process
>
> e.g.
> $110101011_2 = 1 * 2^8 + 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 427_{10}$
>
> $1054_8 = 1 * 8^3 + 0 * 8^2 + 5 * 8^1 + 4 * 8^0 = 556_{10}$
>
> $1AC3_{16} = 1 * 16^3 + 10_{10} * 16^2 + 12_{10} * 16^1 + 3 * 16^0 = 6851_{10}$

So, you might wonder: *How can it help us with fractions*? And I can answer: we present fraction part as some power of $base_x$! Consider following example:

$$100.75_{10} = \begin{cases} 01100100_2 \text{— integer part} \\ 11000000_2 \text{— fraction part} \end{cases}$$

> *Notice, we can trim all leading zeros in* integer part *and all trailing zeros in the* fraction part *without affecting a value. We are showing*

[24]There are a number of ways to deal with fractions. Example, described here is subtype of fixed-point fractional numbers. This particular method of dealing with them was chosen due to being somewhat simple

*those zeros here for formatting purposes and to explicitly show, that we allocated one byte for each part*[25].

We already know, how we got an *integer part* of our number. But how on Earth did we calculate *fraction part*?

To calculate a fraction part in binary, we should perform following operations:

1. Trim integer part from our number, *we should use only fraction part.*
2. Multiply fraction part by 2. Store resulting *integer part* somewhere.
3. Repeat[26] steps 1–2, passing result from the second step to the first step.
4. List of *integer* parts is your binary number.

e.g.

| Step # | Number | Result | Result's integer part |
|---|---|---|---|
| 1 | 0.75 -> 0.75 | 0.75 * 2 = 1.5 | 1 |
| 2 | 1.5 -> 0.5 | 0.5 * 2 = 1.0 | 1 |

So, $11_2$ is our result for *fraction part.*

We can, for clarification purposes, show our result as: $100.75_{10} = 1100100.11_2$. Our transformation method would still work[27]:

$$100.75_{10} = 1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2}$$

That's how we can store fractional numbers, for example. Although, it might not usually be the only and exact case how it's done, but it gives us nice insight into how we can use only binary integer number to store something not binary and not integer. We must also consider the fact, that not every number composes so nicely into sum of $2^{-x}$. *We cannot store* $3.03_{10}$*, for example, this way without some sort of rounding.* It may give insight into why computers sometimes act weird on calculations (computers have trouble computing, isn't it nice?).

[25] It's not imperative, we could allocate any number of bits/bytes we wanted

[26] It is possible to be stuck in this loop forever, so you should cap maximum repeatition times (maximum number of digits after the dot).

[27] We could omit multiplications of zero, without affecting the value. They are shown for clarity

### 2.3.4 Did I waste your time?

However, the main point of this section is not just to give you info on how to transform different numbers in different bases. It's so basic and fundamental operation that there is practically no way that you won't find an instrument to do this. In thousands of systems and programming languages, most of the time, you wouldn't even bother with those operations. At most, you would just write something along the lines 'Hey, transform this number to binary for me, will ya?' *So why did I wasted your time on this section*? Well, *it is fundamental, nonetheless*. Practically everything is built on the principles that we talked about here. And the main idea of this section is not about some information transformation details — it's that in general, *any information with one way or another will be transformed into binary form at some point*, if you are using a computer. I just thought it would be nice to give a couple of examples for it not to sound magic-y of sorts

Main question of this section was: *Do we really need to invent new encoding formats or some types of data*? And now, I want to believe, I can give an answer, that can be understood, with the background consisting only of this section. *Yes and no*. There wouldn't be any point in programming itself, if we wouldn't inventing something new with said program, would it? The main goal of programming is to *teach your computer doing something new*. So, you will create something new, at least for your computer. But, doing so, *it would be absolutely dreadful to re-invent everything*. So, you will use some standards and wheels that were invented before you. It's only natural. If I wanted to, say, create a video-player, I would love to (I hope) create some player-specific functionality. However, I wouldn't want to explain to my player what the number is. Or what is text. Or how to understand what time is it. *I need to teach my player use it*, of course, but *it doesn't mean I need to re-invent it* for my player.

It's also interesting to observe, that nowadays programmers, in general, shouldn't bother with this stuff. I mean, they should know it, of course, but it's possible to be able to program something and not to get into details of how this works. There are of course some fields, where doing these operations from scratch is a must[28] but it's not really that widespread anymore. Programming as a profession and as a science[29] is somewhat new, of course. However, it's not as new as the most people think. I'd like to dwell on this subject in the next section. All I wanted to say now, is that computers now are way-way more powerful than before. It allows us to automate many things, even automation itself. Some time before *every* programmer had to know *exactly* how things like that worked on *his exact computer*. There just weren't as many computers, as many instruments and performance capacity as today. Today, for better or worse, *we have the luxury of not dwelling into much detail like we did in this section*. It's like we are now the managers of computer, when we only say: 'do that, take that, change those' and expect result, without having to explain in great deal of detail *how it must be done exactly*. It is a relative thing, of course. We still need

[28] Somebody did write those instruments we all use, didn't he?

[29] Computer Science is a more correct term here

to explain many things to computer, keeping in mind many intricate details. However, *with every generation of technology, we tend to explain to computers less* '*How*' *and more* '*What*'.

## 2.4 Countess, weaving patterns and count machines

### 2.4.1 Ancient calculators

So, essentially, what is a computer? And yes, there is a definition from Merriam-Webster dictionary in the first pages. *But what does computer do, in its essence*?. Well, it *computes*, duh. *So, the computer is a buffed-up calculator*? *Well, it is an oversimplification, but not really untrue.* If a five-year old asked me what a computer is, I would definetely tell him, it is a calculator, but very powerful and calculating all sorts of stuff[30].

I want to devote this section to a brief historical overview of computers. Please, keep in mind, that it is a *vast* topic. However, there are some moments in the very beginning of computer history, that interests me the most, regarding this essay. Since this essay puts the programming perspective into focus, I want to explore a bit, how did the programming started.

But where should we start? Well, since computer was oversimplied to rather big calculator, I suggest we start with the calculators.

The first attempts to help us count things are ancient. The first thing coming to mind is an *abacus* or *abaci* for plural. We don't know exacly when or where it emerged[31]. However, we do know, that many cultures used some form of abacus or counting frame for calculating purposes. The principle behind those auxiliary devices they used is virtually the same in every cultures, despite the details being a little bit different. We have multiple names for different types of abaci in different cultures[32]:

| Original name | Transcription | Origin | Additional Info |
|---|---|---|---|
| 算盘 | suanpan | China | |
| そろばん | soroban | Japan | Can be represented as '算盤' also |
| 주판 | jupan | Korea | Could also be called supan (수판), jusan (주산) |
| nepohualtzintzin | | Aztec Culture | |
| с ч е т ы | schoty | Russia | |

Table 2.9: Abaci in different cultures

[30] I want to believe, that an explanation in the previous section gave you a clue, how it transforms all the stuff into numbers.

[31] https://en.wikipedia.org/wiki/Abacus

[32] https://en.wikipedia.org/wiki/Abacus

However, those are purely mechanical and manual devices, main principle of which is very similar to how we convert numbers to different bases. They still require us to 'translate' our numbers to abacus system and, after we've done our math, translate them back.

Figure 2.1: Blaise Pascal

The first succesful automation attempt is attributed to Blaise Pascal, with his *Arithmetic Machine* which is also called a *Pascaline*. It was designed and built between 1642 and 1644[33]. Pascaline could only add and subtract numbers, using rotational dials as an input.

There are several pascalines still intact nowadays, most of the remaining ones are in european museums. Being the first calculator isn't the only achievement of pascaline. Pascal was only 18 years old, when he started designing his machine, trying to help his father with accounting. He went through about 50 prototypes before settled on the final one. Later Pascal presented his machine to the public, and, eventually, to the King of France, receiving a royal privilege, which was basically a patent in those days. And yes — his father did use it in work afterwards

Pascaline wasn't a computer, but it was first in many ways — first calculator, which was afterwards commercialized, used in business and patented. Many of subsequent attemps to farther automate calculations were directly inspired by Pascaline.

### 2.4.2 Luddites' nightmare

The next machine of our interest is not a computer either. It isn't even a calculator — it's a loom. I cannot say much about weaving history, but in computer history, it was indeed the most significant loom there is. In 1804 a french weaver and merchant, Joseph Marie Jacquard invented a *Jacquard Machine*.

Figure 2.2: Joseph Marie Jacquard

Beginning of the 19$^{th}$ century is pretty much a middle of industrial revolution. New fancy industrial looms are practically a symbol of the new era. Industrial revolution changed our world forever, marking a *transition from hand production methods to machines*. Despite Great Britain being the origin of the industrial revolution, it spread eventually over the continent, and afterwards, the world. Industrial revolution eventually lead to the *emergence of the capitalist economy*. In other words — *Industrial revolition was a big deal*.

[33]https://www.britannica.com/technology/Pascaline

So, amidst all this revolution going, we could see how a work, that was being done by 100 men before can be done by 10 men with machines. It was *super effecient*. Inventors tried to find a way to increase production effeciency (and therefore revenue) by exploring technical capabilities of the machines they worked with. So Joseph Jacquard invented his machine, which was basically an *attachment to the industrial loom*. A loom with an attached machine was subsequently called *Jacquard Loom*[34].

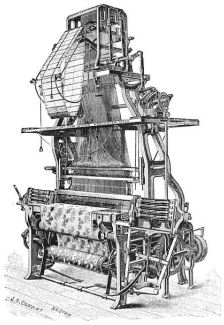

Figure 2.3: Jacquard Loom

The main purpose of Jacquard's attachment was an *pattern weaving automation*. It used a chain of special cards laced together in a continuous, looped sequence. Those cards will later be called *punch cards*. Punch card is, essentially a card, with designated spaces for holes. Later, some designated space would be 'punched' to produce a hole in area. Some spaces were omitted during 'punching'. After 'punching' those holes we would have a card, partially filled with holes. Jacquard Loom relies on a simple mechanical action of those cards. While continuously moving, those cards would move controlling rods of the loom, therefore affect knot being weaved. If there was no hole in the area, rod will move since card will push it. If there was a hole, rod wouldn't move since it would go through hole.

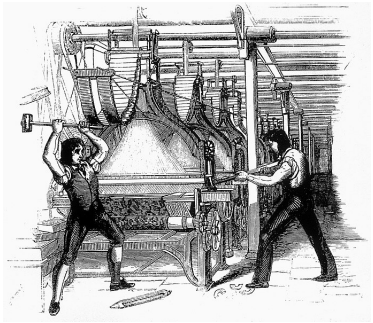

Figure 2.4: 1844's depiction of Luddites destroying the loom

This machine would *drastically* impact effeciency, since it was no longer required to be of high skill to weave complicate patterns. It, essentially, gave manufacturers ability to *store design patterns and to reproduce them indefinitely*[35]. The man behind the punching process still needed to be trained and qualified, but to just reproduce a readily punched design wasn't much of an issue even to a low-skill worker. Just imagine a reaction from weavers, that used to work by hand. No surprise, that *Luddites* [36] destroyed such machines to protest against their usage.

[34]Jacquard Loom is a general term, which does not describe any concrete loom, but rather any loom with the control mechanism, allowing pattern weaving automation

[35]At least until punch cards are in fine condition

[36]The term itself initially related to an organisation of English textile workers. However, over time it has come to mean one opposed to new technologies in general

### 2.4.3 Steam, calculators and British Government

Figure 2.5: Charles Babbage

Well, the idea of using automation in weaving patterns have touched deeply not only the Luddites, but also at least one mathematician, which was also a philosopher, inventor and mechanical engineer. It's just so happened this was also a man, who will be considered as the 'father of the computer'[37] by many. Some of his works even touched the subject of industrialization and economy, which influenced Karl Marx [38]. A man called Charles Babbage was deeply inspired by Joseph Jacquard's invention and intended to use his ingenious punch cards in his own machine.

Charles Babbage was an inventor of 2 machines, that are of interest in this essay: *Difference Engine* and *Analytical Engine*. Both of these machines were not finished in his lifetime, unfortunately. Nonetheless, this heritage of 2 unfinished machines marked a signinificant point in history of computers. More than that, *those 2 machines gave an opportunity for the first ever programmer to become a first ever programmer*.

The Difference Engine, essentially was a giant mechanical calculator, that was powered by steam and printed results of it's computations in a table. The format of the result was chosen due to being practical in that time — many fields relied on tabular data to perform operations. One of the most notable example is a 'Nautical Almanac', which was crucial for navigation and astronomy[39]. But, constructing such a machine would require a formidable expenses and time. So, Babbage did what any startup nowadays would do — he started seeking for investments. In 1822, he wrote a letter[40] to the President of Royal Society[41], in which he presented a detailed explanation and description of his would-be machine[42]. He also published this letter as pamphlet and sent it to other people, he deemed influential.

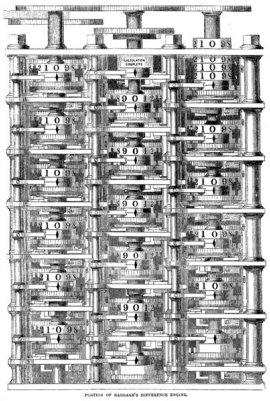

Figure 2.6: Part of Babbage's Difference Engine

[37] https://cse.umn.edu/cbi/who-was-charles-babbage

[38] https://projects.exeter.ac.uk/babbage/rosenb.html

[39] https://www.historyofinformation.com/detail.php?id=418

[40] https://play.google.com/books/reader?id=YBHnAAAAMAAJ&pg=GBS.PP2&printsec=frontcover&output=reader&hl=en

[41] Organization that exists to promote academic disciplines and science.

[42] https://www.historyofinformation.com/detail.php?id=450

One copy of this letter did reach a Lord of Treasury, who referred it to the Royal Society. After receiveing an endorsement letter and favorable report, Treasury decided to invest in Babbage's invention[43]. Project ended up being at least 17 times more costly tnan initial estimate, not being finished at all, by the end of the next decade, eventually foundering in 1833. Babbage was initially granted a £1000, but for the next decade claimed a total sum of £17,000 of goverment money. It can be roughly estimated to £2,150,000[44] (around $2,836,000) in 2022 money.

In 1833 Charles Babbage threw a party, where he demostrated his guests, mostly members of high society, a part the Difference Engine. There were many guests that night, but we are interested in one in particular. One of the attendees was Lady Ada Byrone[45], the only legitimate child of Lord Byron, an English Poet. She showed an interest in this machine and became a life-long friend of Charles Babbage. Correspondence between the two is pretty well preserved to this day, providing insight to the next Babbage's invention.

As was mentioned before, Difference Engine project did end up exceeding given funding. Charles Babbage wasn't able to continue to work on his machine — he couldn't afford his chief engineer in the project, Joseph Clement[46]. This has lead Babbage to attempt an even more ambitious project in 1834 — *Analytical Engine*. While working on the Difference Engine, Babbage began to imagine ways of improving it, for it to support other kinds of calculations. Analytical Engine was something, we could call a computer in a general sense. It could be automated to perform any calculation set before it programmatically[47]. I'm jumping a little bit forward here, but *in the Babbage's lifetime, only a small trial engine was constructed*.

Figure 2.7: Punch cards used to program the machine
Photo by Karoly Lorentey, sourced from wikimedia commons, under Creative Commons Attribution 2.0 Generic license.

Analytical Engine was designed to consist of four major elements: the mill, the store, the reader and the printer. Functionally, modern computers pretty much repeat this architecture. Those elements have very much resemblance with somewhat modern computer architectures. In this architecture, the mill acts as a 'brain' of analytical engine, performing mathematical computations. The store was a place, where computer would hold it's data prior to computation. The reader and printer were an input and output units — to 'communicate' with an outside world, allowing to consume and output information. All computers now pretty much have the same components

[43]https://www.historyofinformation.com/detail.php?id=450

[44]https://www.in2013dollars.com/uk/inflation/1833?amount=17000

[45]She later will be known as Countess Ada Lovelace

[46]Clement did end up owning all work he've done to himself, leaving Babbage virtually empty-handed

[47]https://www.britannica.com/technology/Analytical-Engine

functionally. If we are talking in informatics term, these components show strong resemblance to von Neumann architecture, described in 1945(!), more than 100 years later.

So, that's where an inspiration from Joseph Jacquard really kicks in! The principle behind punch cards, used in Jacquard Loom was adapted by Babbage to input data into his computer! It seems, that Babbage deeply respected Jacquard, having his woven portrait (with the help of Jacquard Loom, of course) in his posession with at least 24,000 punch cards used to create it[48].

So, operating with those cards would give to one an ability to code necessary instructions for Analytical Engine to follow. Having a *store* component, where commands could be held before processing gave an ability to perform operations *out of order*. In other words, computer was able not to just blindly follow commands punched on those cards, one by one, but rather have some 'jumping' around in case of necessity. This opens a whole new level of controlling the program 'flow', allowing for complex behaviour, based on meeting some criteria. 'If that then do this' kind of stuff. Just to note, this feature was missing in many of the early computers of the 20$^{\text{th}}$ century[49]!

### 2.4.4 Enchantress of Numbers and Italy's Prime-Minister

Using punch cards as a format of input data not only have fascinated Babbage, but Ada Byrone too. For the next ten years, after the aforementioned party, she did study almost daily, learning from Babbage all she could about the machine. In the meantime she married William King in 1835, thus making her Ada King. In 1838 William King was made Earl of Lovelace, making Ada the Countess of Lovelace. *She mostly remembered by the name Ada Lovelace.* The two worked very closely, corresponding on the regular basis.

Babbage gave lectures about his inventions sometimes. On one of such occasions he had a very special listener — *Luigi Federico Menabrea*, military student. Menabrea had quite a lot of achievements in his life: he later became engineer, doctor of mathematics, general, Count, Marquess of Valdora and seventh prime minister of Italy. However, we are more interested in his academic publications. After learning about Babbage's Analytical Engine, he wrote a paper 'Notions sur la machine analytique de M. Charles Babbage'[50] in which he, with great detail, described mathematical nuances regarding Babbage's machine. He also written it in French, as you could pick up from the title.

Ada decided to translate Menabrea's work in English, titled 'Sketch of the Analytical Engine invented by Charles Babbage'. Ada not only translated this publication, but also *extended it with her own thought and commentary.* So, it became 'Sketch of the Analytical Engine invented by Charles Babbage with Notes by the Translator Augusta Ada King, Countess Lovelace'. Her resulting

[48] https://www.sciencehistory.org/distillations/magazine/the-french-connection

[49] https://www.britannica.com/technology/Analytical-Engine

[50] http://www.bibnum.education.fr/calcul-informatique/calcul/notions-sur-la-machine-analytique-de-m-charles-babbage

work was published in 1843 and *3 times as long as the original paper*[51].

She also illustrated in her notes a sequential solution of various problems, through input in a form of punch cards into Analytical Engine. So, basically, she used those punch cards the same way we are using keyboard and mouse to interact with the machine. Thus, making it the *first ever documented and published program — a set of instructions for computer to follow.* So, meet the first programmer ever — Countess Lovelace.

Figure 2.8:
Augusta Ada King,
Countess of Lovelace

Although there are some disputes regarding the title of the 'first programmer ever'[52], there is one thing virtually no one disputes: *her understanding of the potential computers have.* Where Babbage have seen his own invention only crunching symbols and numbers, *Ada saw potential to analyze and work with anything, provided there is a code for it, machine could understand.* In her notes for the translation, she wrote about objects besides numbers to be expressed by 'science of operations' and gave an example of possibility for sounds to be expressed in such a way[53]. Ada have also questioned computer's ability to 'think on its own', and gave her thoughts on what will be later called *artificial intelligence*[54].

However, the history of Analytical Engine have ended due to a lack of funding, and it remained mostly on paper for the *centuries* to come. Ada's late life have taken a rather grim end, with lots of gambling, debts, opioid medications, adultery rumours and eventual death in 1852, 9 years after her first program and only in the of 36 years old[55].

Charles Babbage continued to work on his machine until his death in 1871. As was said, machine was never finished in his lifetime. A version of his *Difference Machine* without printing mechanism was built in 1991 by London Science Museum[56]. Constructors tried to adjust for physical properties achievable in 19$^{\text{th}}$ century. It was concluded, that machine would've worked[57].

### 2.4.5 Brief timeline of the computer history

Computer history is filled with many significant occasions since 19$^{\text{th}}$ century. As was said, it is indeed a vast topic, although we will mention few[58]:

[51] https://www.historyofinformation.com/detail.php?id=467
[52] Most sources still say it is Ada Lovelace
[53] https://www.historyofinformation.com/detail.php?id=467
[54] https://medium.com/swlh/ada-lovelace-her-objection-e189717bd262
[55] https://www.biography.com/scholar/ada-lovelace
[56] https://collection.sciencemuseumgroup.org.uk/objects/co526657/difference-engine-no-2-designed-by-charles-babbage-built-by-science-museum-difference-engine
[57] https://en.wikipedia.org/wiki/Difference_engine
[58] Most occasions here are sourced from https://www.computerhistory.org/timeline/

| Year | Details |
|---|---|
| 1941 | Konrad Zuse finishes first programmable, electric digital computer, called Z3. Basically, all the Babbage wanted, but with electricity in the form, we most familiar with, today |
| 1945 | John von Neumann described an architecture, that is the basis for virtually all of the computers we have today. This architecture later will be known as 'von Neumann Architecture' or 'Architecture of von Neumann Machine' |
| 1948 | Claude Shannon formulated his first thoughts regarding what will be regarded as 'Information Theory' in the future |
| | First computer program ran on the computer |
| 1952 | Grace Hopper invented first high-level programming language, A-0. It will evolve into COBOL later. |
| 1956 | Keyboard was succesfully connected to computer. Prior to this point, all programming had to be done by punch cards or similar non-keyboard altertnatives |
| 1957 | FORTRAN created, *first widely used high-level language.* |
| 1958 | LISP developed |
| 1960 | COBOL developed. COBOL *still* highly in use today, especially in financial sector. Up to 80% of the world's daily business transaction rely on COBOL, source: https://www.ibm.com/blogs/ibm-training/did-you-know-80-percent-of-the-worlds-daily-business-transactions-rely-on-cobol/ |
| | ALGOL-60 developed |
| 1969 | ARPANET first online. ARPANET is a direct predecessor for the *Internet* we know today |
| | Kenneth Thompson and Dennis Ritchie developed UNIX. It's not too much of a stretch to say, that it is one of the most significant OS in history of computers. They influenced virtually all OS in use today. Because of the UNIX, or with it heavy influence, we got Linux, iOS, OS X, Android and Windows |
| 1970 | First ATM can be used by general public. |
| | Pascal programming language developed. |

| | |
|---|---|
| 1972 | C language developed. C is one of the *most influential* programming language there is. It's average popularity, by TIOBE index, consistently ranked as *top-2 since 1987*, source: https://www.tiobe.com/tiobe-index/. TIOBE index is a measure of programming languages, created and maintained by company of the same name in Netherlands. |
| 1973 | First handheld cellular mobile phone invented. Start of a mobile network we know today. |
| | First successful inter-network communications. Birth of the Internet as we know it today. |
| 1977 | Apple II computer is developed by Steve Wozniak and marketed by Steve Jobs. One of the first computers, intended for personal use. |
| | Atari video game console released. One of the first game consoles in history |
| 1978 | First Multi-User Domain games appeared. They allowed for multiple players play against each other. One of the first multiplayer competitive game experinces. |
| | First Computer Worm created. |
| 1983 | Internet officially launched |
| | Microsoft Word officially launched |
| 1985 | C++ released |
| 1987 | Basic parameters for GSM standard agreed. GSM is the main standard used in mobile network today. |
| | Perl developed |
| 1990 | First commercially succesful Windows OS version released, Windows 3.0 |
| | Photoshop initially released |
| 1991 | Linus Torvalds released Linux kernel |
| | Charles Babbage's Difference Engine #2 is constructed at London Museum. |
| 1993 | First online ads appeared |
| 1995 | Java 1.0 introduced |
| | Javascript released |
| 1999 | WiFi routers started to gain popularity for in-home usage. |
| 2000 | First mobile phone with camera released |
| | USB flash drives first apperaed on the market. |

| | |
|---|---|
| | Y2K problem. 'Y2K' refers to potential computer errors related to the formatting and storage of calendar data. Not every program validly operated with dates and there were many speculations about how this would affect computers. |
| 2004 | World of Warcraft comes on-line |
| | Google's IPO |

Table 2.10: Some of significant occasions in computer history

So, although this table cannot possibly reflect all significant events in the computer history, it demontsrates nicely how immensely far we've gone from the Babbage's steam-powered machines and Pascal's mechanical calculators.

### 2.4.6 History matters

I occasionally hear an argument about whether or not it is necessary to learn something as obscure as the history of computers and those weird looms and whatnot. The main reasoning of such argument being that it's not necessary to know history of programming to learn it. Although it might be true, nonetheless I don't only think it wouldn't hurt too much to know it, I am sure that *knowing such details could give a real insight to some technologies we use today.* Computer Science may not be the most ancient of sciences, but it does have it's history. It's not the history of the machines we are interested in, no. We are interested in the history of the *human thought.* It's not the machine that builds itself — it's human's work. *Any scientific and applicable discipline we know today is a result of accumulation of said thoughts.*

Today we can see number of technologies, that emerged seemingly out of nowhere and dominated market, science, and most importantly — human minds. Just an example: blockchain and cryptocurrencies that today gracefully entered mainstream news and media were still a geeks' venue in early 2010s. But even if you knew about Bitcoin, in the year when Satoshi Nakamoto published his paper[59], did you knew, that blockchain was firstly described not in 2009, but in 1991(!)[60], 18 years before the notorius use case example in the face of the Bitcoin emerged? 18 years — is a considerate time-span, especially in IT. In IT terms, to study such an old documents is the equivalent of archaeological studies[61]. First works on the decentralised digital currency are emerged in 1998. Just to put it differently: *in the end of the last century*, which is crazy to hear, talking about IT and something that entered mainstream today.

Or we can see another example of the technology dominant today, which roots are deeply in the past: Artificial Intelligence and Machine Learning. Face

[59] https://bitcoin.org/bitcoin.pdf

[60] https://www.icaew.com/technical/technology/blockchain-and-cryptoassets/blockchain-articles/what-is-blockchain/history

[61] Maybe a bit of exaggeration, but you get the idea

recognitions, self-driving cars, fraud detections. AI and ML in some form or another is pretty much in the mainstream now. *The field of artificial intelligence research was founded as an academic discipline in 1956*[62], around 66(!) years ago from the moment I am writing this essay. It doen't sound like a bleeding edge of a technology now, eh[63]?

Oftentimes what we call 'modern' and 'bleeding edge' is a simple reincarnation of old ideas. It's also not due to plagiarism or stealing, but simple practicality. Some concepts were just not implementable at the time of emerging. It mostly occurs due to hardware limitations. First self-driving car, using neural networks[64] was created in 1989[65]. It was fairy simple by today's standards, but it served as a proof-of-concept, proving feasibility of further research.

For us to delve in history — is to understand motives and thoughts of our fellow curious minds. It helps us navigate and understand fundamental reasons of why something was made the way it was. Sometimes, revisiting old concepts can lead us to something new with the help of capabilities, provided by modern technologies.

[62]https://www.cantorsparadise.com/the-birthplace-of-ai-9ab7d4e5fb00?gi=a4c19646195c
[63]AI and ML has undoubtedly came a long way since then. Still though.
[64]Technology imitating neurons in our brain, mostly used in ML and AI disciplines
[65]https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn

## 2.5 Modern computers

### 2.5.1 Hardware every computer needs

This section is mostly devoted to computers as we know them today. We are surrounded by computers. Our PCs[66], laptops, tablets, mobile phones are computers. What do they have in common?

Well, first of all — they all have different *hardware*.

> Hardware — is all of the computer's tangible parts or components. Pretty much everything that you can see, once you disassemble your device is hardware.

Hardware is the basis of any computer. Once you got limited by hardware, there is not much you can do without changing that hardware. Hardware is like the lowest baseline measurement of what computer can do for you. To provide an analogy with a car: if your engine physically is not capable to get you 100 km/h in 4 seconds, and you absolutely need to go that fast, well... change either the engine or the car itself[67].

So, *hardware often acts as a physical limit of what your computer can do*. If you absolutely need to save 100 GB of photos on your computer and you just don't have any — good luck. You can't just 'install more disk', to get more space, you need to open up your computer case and install new disk physically. You can always try to somehow 'shrink' photos, but that is entirely different story.

> You cannot change hardware without physically installing/reinstalling some computer component. It cannot be done with some command or programm, it's just screwdriver and you, buddy

So, what hardware every computer needs? Well, maybe not every little one of them, but the majority? What will be described here sticks well with most PCs, laptops, phones and tablets. There are, of course, all sorts of different other computers, but, for the most cases, hardware, described here, can be adapted to them (logically) with very minor tweaks.

1. CPU[68]
2. PSU[69]
3. RAM[70]
4. Persistent Memory storage

[66] Personal Computers

[67] To change the car here means one of two thing: change the car entirely or change something in its configuration to get that fast, e.g. get it to be lighter

[68] Central Processing Unit

[69] Power Supply Unit

[70] Rapid Access Memory

5. GPU[71]

6. Motherboard

And finally, it all should be mounted and secured in some case. Usually end-user interacts and associate hardware with case.

The list of possible hardware some computer may utilize in some form is just enormous. It makes no sense to try and describe all possible parts there is. However, the list above is pretty comprehensive in a sense of a bare minimum for computer to function and for user to be able to interact with it.

> It's worth noting, that each and every one piece of hardware described here is a complex technological device. To understand in details how works even one of those components — means to research a lot of specialized literature and be capable to comprehend a lot of engineering disciplines such as electrical engineering and digital circuit design. This essay is intended only to introduce general functional capabilities of those devices, without dwelling too much into details, that a layman would find hard to understand.

### 2.5.2 CPU

As was said before — CPU is a 'brain' of the computer. In the end, all operations, that somehow process any data is executed on CPU level. CPU is able to execute instructions, written in *machine code*, presented in a *binary*[72] format. Such instructions can manipulate memory addresses (to allocate some space in memory, or load something from it), perform arithmetic and comparative operations on operands[73] (usually with one or two operands), perform logic operations [74], control flow instructions (changing order of how program is executed) and other fundamental operations.

So... that's it? To learn programming — means simply to learn those machine code instructions and find some way to shove it into CPU for it to execute? Well... while this would most certainly make you a programmer, there are deep caveats here. This question is addressed further in the chapter 'Why Computer?' on 37

### 2.5.3 Memory

Memory, regadless of the specific type we are talking about, serves one thing: *to act as a storage*. There is, mainly, two types of memory in a computer[75]:

[71] Graphical Processing Unit

[72] Presented in $base_2$

[73] *Operand* is an object of some *operation*. e.g. in '5 + 2', '+' — is an operation, '5' and '2' — operands

[74] Special operations such as 'AND', 'OR', 'XOR' and others. They are subject of the discipline called 'Boolean Algebra'.

[75] CPU and GPU usually have their own memory. However, they are not of particular interest in this regard here

1. Volatile
2. Non-volatile

Main functional distinction between them: *volatile memory* cannot save something if there is no electric power. In other words — *it empties once, the computer been turned off*. On the other hand — *non-volatile memory* is *durable*, meaning it can store something even in case of power been cut off.

Volatile memory is usually associated with RAM. Basically it's like a literal *cash register*, where company stores something, to give out change and to which it adds today's revenue. It's accessible, fast and handy, but you wouldn't store something more substantial than one day's revenue — it simply wouldn't fit. And if cashier vanishes with the register — only today's revenue lost.

Non-volatile memory is usually associated with some sort of drive. In most cases it's HDD[76]- and SSD[77]-like structures. It's *the thing* that we constantly running out trying to save all the photos on our phones and laptops, or when we install a bunch of programs on our computers. To continue our association — it's like a *safe deposit box*, or a *bank vault*. It's secure and suitable for long-term storage. Disaster can still happen, of course, but it's magnitude must be considerate for the bank to fail us, opposite to cash register scenario. However it's a bit of a hustle to go and put something in a said *vault*. You wouldn't go to it every 5 minutes to give out change from 5$, would you? No, you would rather *accumulate* substantial amount in your cash register first, then go to the bank and deposit it.

*This simple analogy sums up why we have 2 types of memory: volatile is fast, but small and not much stable and non-volatile is stable and large, but slow. We combine them to achieve what we consider optimal performance.*

### 2.5.4 GPU

GPU's primary function is to render images on screen. They can be oversimplified to the 'special processors for rendering images'. Most computers that are accessible by the end-user usually have some output device, where information is displayed. Usually it is some type of screen. We have screens on our laptops, mobile phones, tablets. For out PCs we usually buy an additional hardware: 'computer monitor' to be able to actually use the computer and see results of our actions on screen.

However, it's not always the case: *computers can be used even without screens*. If you don't need to display your information on screen — you don't need the GPU.

### 2.5.5 Motherboard and PSU

Motherboard handles all communications between all the hardware attached to your computer. Via motherboard electricity travels, 'telling' other components

[76]Hard Disk Drive
[77]Solid State Drive

what to do in binary format.

Every computer needs power — whether it's AC (power outlet) or DC (batteries).

# Chapter 3

# Why Computer?

## 3.1 Can we talk about programming now?

### 3.1.1 What is software?

So, we discussed briefly what information is, short history of computers and what hardware is. But how does it all interact with each other to give us experience we used to call normal computer work?

To put it simply — hardware is the bare bones, blood system, muscles and joints of the computer. However, it's just laying there, doing nothing, *until we tell it what to do*. Where does the impulse come from? How can we step up from all the components just assembled together to interactions between them, producing some useful work for us? *Well, we must program it.*

Let's say, that you want to view some photos, saved on your disk in your laptop. What actions are you gonna do to view those photos? I imagine something similar to this:

1. Turn the power on
2. Wait for display to show something
3. See welcome screen, once your OS finished booting up
4. Enter your password to log in
5. Locate your photos somewhere on computer
6. Click on it, for application to appear and show your image

*Everything in this list, except for turning the power on relied on some software to work correctly.* It is software, that displaying things on your screen. It is software searches for OS files to boot up computer, it is software transforming your keyboard typing to something computer can understand, it is software waits for the correct password to be entered, it is software shows you neat folders and

files for you to navigate, it is software starting application to show you photos. There is a *ton of software involved* in such a simple process, isn't it?

We also must answer for a crucial question: 'what is software?'. Well, it's instructions, telling the computer what to do[1]. Software is just a fancy term for computer programs.

### 3.1.2 Behold! An assembly language

So, how does one tell computer what to do? Well, mostly, it boils down to tell *processor* what to do[2]. As was discussed in 'Modern computers' on page 33, processors are able to receive and implement instructions in *machine code*. Machine code is always represented in binary format, meaning it's just 'raw' zeros and ones. It's very inconvenient for human to send something in binary format — too cumbersome and error-prone.

To overcome this nuisance, we must create a 'translator' of sorts — for us to write in some way, comfortable enough for human and to translate it into said machine code which is so convenient for the machines. *Thus, we firstly created a distinction with something conventient for humans and for the machines.* Anyhow, one of the first such *abstractions* over machine code is an *assembly language*. Main purpose of assembly language is to translate *as directly as possible* to machine code instructions of the processor. Instead of being raw binary — it is actually a *language*, in this context, meaning that it can be written as text. Just to illustrate what it may look like, we can look at MS-DOS sources[3]. Code presented above can be found in file 'PRINT.ASM'[4] in MS-DOS repository.

```
;Interrupt routines
ASSUME  CS:DG,DS:NOTHING,ES:NOTHING,SS:NOTHING
        IF      HARDINT
HDSPINT:                        ;Hardware interrupt entry point
        INC     [TICKCNT]       ;Tick
        INC     [TICKSUB]       ;Tick
        CMP     [SLICECNT],0
        JZ      TIMENOW
        DEC     [SLICECNT]      ;Count down
        JMP     SHORT CHAININT  ;Not time yet
TIMENOW:
        CMP     [BUSY],0        ;See if interrupting ourself
        JNZ     CHAININT
        PUSH    DS
        PUSH    SI
```

[1] https://www.britannica.com/technology/software

[2] I purposefully don't write 'CPU' instead of processor here since some programs run on GPU processors

[3] https://github.com/microsoft/MS-DOS

[4] https://github.com/microsoft/MS-DOS/blob/master/v2.0/source/PRINT.ASM

```
        LDS     SI,[INDOS]      ;Check for making DOS calls
        CMP     BYTE PTR [SI],0
        POP     SI
        POP     DS
        JNZ     CHAINT          ;DOS is Buisy
        INC     [BUSY]          ;Exclude furthur interrupts
        MOV     [TICKCNT],0     ;Reset tick counter
        MOV     [TICKSUB],0     ;Reset tick counter
        STI                     ;Keep things rolling

        IF      AINT
        MOV     AL,EOI          ;Acknowledge interrupt
        OUT     AKPORT,AL
        ENDIF

        CALL    DOINT
        CLI
        MOV     [SLICECNT],TIMESLICE    ;Either soft or hard int resets time slice
        MOV     [BUSY],0        ;Done, let others in
CHAINT:
        JMP     [NEXTINT]       ;Chain to next clock routine
        ENDIF


SPINT:                          ;INT 28H entry point
        IF      HARDINT
        CMP     [BUSY],0
        JNZ     NXTSP
        INC     [BUSY]          ;Exclude hardware interrupt
        INC     [SOFINT]        ;Indicate a software int in progress
        ENDIF

        STI                     ;Hardware interrupts ok on INT 28H entry
        CALL    DOINT

        IF      HARDINT
        CLI
        MOV     [SOFINT],0      ;Indicate INT done
        MOV     [SLICECNT],TIMESLICE    ;Either soft or hard int resets time slice
        MOV     [BUSY],0
        ENDIF

NXTSP:  JMP     [SPNEXT]        ;Chain to next INT 28

DOINT:
        PUSH    SI
```

```
        MOV     SI,[CURRFIL]
        INC     SI
        INC     SI
        CMP     BYTE PTR CS:[SI],-1
        POP     SI
        JNZ     GOAHEAD
        JMP     SPRET           ;Nothing to do
GOAHEAD:
        PUSH    AX              ;Need a working register
        MOV     [SSsave],SS
        MOV     [SPsave],SP
        MOV     AX,CS
        CLI
;Go to internal stack to prevent INT 24 overflowing system stack
        MOV     SS,AX
        MOV     SP,OFFSET DG:ISTACK
        STI
        PUSH    ES
        PUSH    DS
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    CS
        POP     DS
```

Don't worry if you don't understand a thing — me neither[5]. Main idea of this demonstration is to show that 'convenient for humans' is a bit of a stretch regarding design of the assembly language. It's just not the case for the overwhelming part of programmers worldwide, who can decipher those instructions *a little* more than total laymen. How come?

Well, although to write in assembly language — is the next level of coolness in my eyes, it just isn't necessary most of the times. Some time ago, indeed, all the programmers had to write in some sort of assembly language to program anything, but those times *mostly* far from now. Why? Well, have you seen an assembly language? It is scary first of all! But *it is a matter of habit* — anyone could learn it and get used to it. Hey, if it get things done, why not? But there is at least one far more fundamental issue with assembly language: did you notice how I said 'programmers had to write in *some sort* of assembly language' a couple of sentences ago? It's not a figure of speech — *there are many kinds of assembly language*.

Since assembly *languages* designed to be as close to the underlying machine code as possible, it's design depends heavily on said machine code. Well, who

[5]My apologies for assembly language teachers in university

defines machine code? *Processor's vendor does.* Processors vary greatly in many ways and *there is just no standard, that can unionize them all under some standardised machine code.* In a practical sense, it means that everything you write and test on your computer may just not be able to work on some other computer if it uses different processor. Well, as we discussed in section 'Simplification through Standartisation', no standartisaion = no scalability.

> Despite all of the issues discussed here, programmers still use 'pure' assembly language sometimes. It is absolutely crucial if you work on something that must use some hardware-specific features or you just must have total control over some hardware-specific behaviour. It is also sometimes a matter of performance — since assembly language is practically one step before machine code, program written in it have the minimal amount of 'intermediaries', which can potentially boost it's performance.

### 3.1.3 Lost in translation

So, wouldn't it be nice if we did have some 'portable' programs, which can work if not completely identically on different processors, but more or less so? Yes, it would be awesome. And, luckily enough, we are not the first people on Earth with such wild dreams. This is a basic idea behind a *compiler* concept.