Introduction to programming

Awesomeness begins

Content

- Programming
- Programming languages
- Choosing a language
- Computer architecture
- Notable ladies in computing
- Appendix

Programming

What is programming?

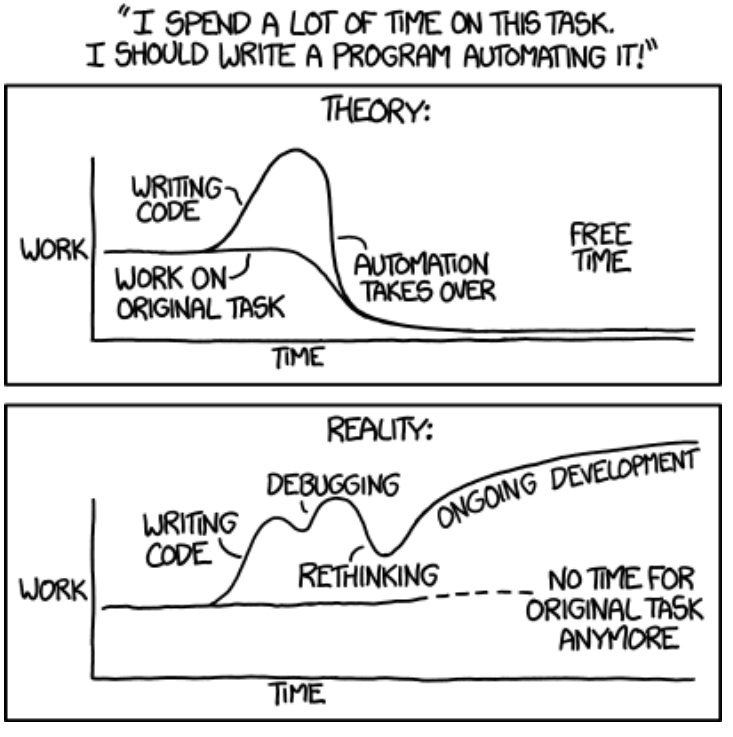
> “ Programming is the act of giving a machine a set of instructions to enable it to perform a certain task.

The computer mindlessly does only and exactly what it's told, provided it's told in a language it understands.

Why do we program?

Computers can do basic computations very fast (much faster than humans, billions per second). That's useful for:

- executing tedious tasks automatically ("booooring, let the machine do it")
  - counting words in a book
  - searching in databases
  - writing personalized email to a thousand people
  - solving sudoku
- solving problems that we know *how* to solve, but we never *actually* solved because it would take more than a lifetime
  - finding the quadrillionth decimal of π (spoiler: it's a 0)
  - enumerating mathematical objects (like trees)
  - cracking cyphers


"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WORK
WRITING CODE
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
WORK
WRITING CODE
DEBUGGING
RETHINKING
ONGOING DEVELOPMENT
NO TIME FOR ORIGINAL TASK ANYMORE
TIME

Human augmentation

Programmable computers are a tool, nothing more, nothing less.

Computers also have interesting input/output devices, which give them access to "non-human" powers, and by extension, gives humans non-human powers:

- video games
- computer generated imagery
- sound synthesis
- networking

Anything you can think of, you can program and *make it happen*. Through language you can make worlds appear and set them in motion.

> I'm a god!!

What can't we program?

Brains are:

- not very fast (a neuron can be activated ~200 times per second)
- very parallel (~100 billion neurons, each connected to ~7000 others)

Computers are:

- extremely fast (~100 billion instructions per second)
- extremely linear (one instruction at a time)

Computers are bad at what brains are good at:

- recognizing shapes in a picture
- understanding/translating natural languages
- doing anything "smart", intuitive, creative, etc...

If you can break up a problem in simple sub-problems, you can write a program to solve it. Otherwise, you'll probably have to cheat (statistical methods, smoke and mirrors...)

Programming languages

Vocabulary

Programming is telling the computer what to do. To "tell" we need to write. To write we need a common language.

- language: syntactic and semantic rules that describe a set of valid textual inputs, and their meaning
- computer language: a language somehow understood by a computer
- programming language: a computer language that describes instructions to be executed by a computer
- (source) code: text written in a computer language
- algorithm: instructions to solve a specific problem (language agnostic)
- program: the implementation of algorithm(s) in a programming language, once compiled or interpreted. "A piece of software". Something that the computer can execute.

Not all computer languages are programming languages (eg. HTML describes data, not instructions).

- to code: to write text in a computer language
- to program: to write text in a *programming* language, ie. to write a program

Programming languages

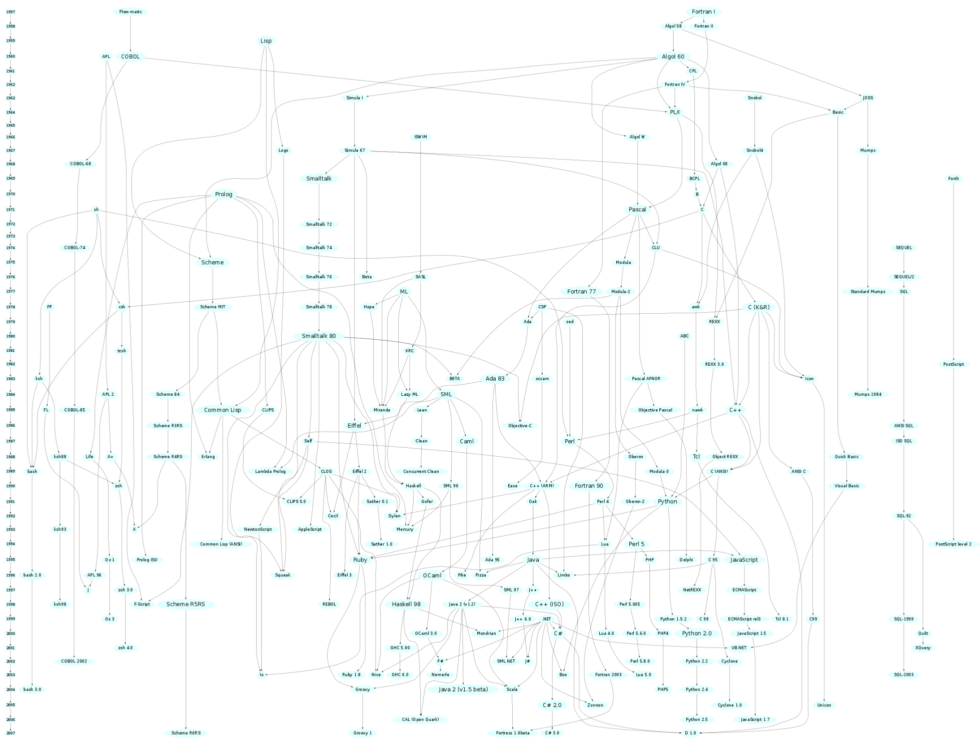
Wikipedia has a list of:

> all notable programming languages in existence, both those in current use and historical ones, in alphabetical order, except for dialects of BASIC and esoteric programming languages.

It's 644 items long. There are many way to categorize and compare these languages. Wikipedia's list of programming languages by category has 42 categories.

You can also sort them by family, which influenced which, which merged to give birth to others, etc.

So, where do we start?



Fortran I
Algol 60
Lisp
COBOL
Simula 67
Smalltalk
Prolog
Pascal
Scheme
ML
Fortran 77
C (K&R)
Smalltalk 80
Ada 83
SML
Common Lisp
Eiffel
C++
Caml
Perl
Tcl
Fortran 90
Python
Perl 5
Ruby
Java
JavaScript
OCaml
Haskell 98
C++ (ISO)
Scheme R5RS
Python 2.0
C#
Java 2 (v1.5 beta)
C# 2.0

General purpose languages

We want to make programs. Native desktop or mobile applications. For that we'll focus on general purpose languages. We can already remove:

- web oriented languages (JavaScript, PHP)
- scientific languages (MATLAB, R, Prolog)
- application specific (VBA)

There are many programming paradigms, or philosophies behind the design of languages. Imperative, functional, object-oriented, concurrent, logic, etc. Not mutually exclusive!

Imperative: statements that change a program state ("do this, do that, if this then do that, etc.")

Low level

One of the many ways to measure and classify languages is "how close to the machine" they are.

At the lowest level:

- machine code (the *only* language a processor actually understands)
- assembly (machine code but in human readable form. Very difficult to write what you mean, very easy to make mistakes. Basically impossible to maintain.)

```
_main:                                  # @main
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %edi
    pushl   %esi
    subl    $8, %esp
    calll   ___main
    movl    $0, (%esp)
    calll   _time
    movl    %eax, (%esp)
    calll   _srand
    movl    $1717986919, %edi       # imm = 0x66666667
```

```
LBB0_1:                                 # %while.body
    calll   _rand
    movl    %eax, %esi
    imull   %edi
    movl    %edx, %eax
    shrl    $31, %eax
    sarl    $3, %edx
    addl    %eax, %edx
    shll    $2, %edx
    leal    (%edx,%edx,4), %eax
    subl    %eax, %esi
    movl    %esi, 4(%esp)
    movl    $L_.str, (%esp)
    calll   _printf
    cmpl    $10, %esi
    jne LBB0_1
# BB#2:                                 # %while.end
    xorl    %eax, %eax
    addl    $8, %esp
    popl    %esi
    popl    %edi
    popl    %ebp
    retl
```

High level

Higher (disguised assembly, plus bonuses to make life easier):

C, C++ (compiled languages, quite verbose, strict syntax, still very lax in terms of memory safety, error prone, but have many higher level constructs that makes coding more "logical" and closer to thought processes)

```c
int main()
{
    int a;
    srand(time(NULL));
    while (1) {
        a = rand() % 20;
        printf("%d\n", a);
        if (a == 10) break;
    }
    return 0;
}
```

Highest level

At the highest level:

Python, Ruby, Lua (interpreted languages, concise syntaxes, very dynamic, they allow to do pretty much anything and it works, memory safe)

```
math.randomseed(os.time())
while true do
    k = math.random(19)
    print(k)
    if k == 10 then break end
end
```

- great, but slower than low level (10-100 times)
- still fast enough for many purposes

Compromise

What's a good compromise?

- new modern languages try to combine efficiency of low-level languages with ease of use of higher level languages (D, Rust, Go). Still a work in progress.
- big projects combine and use different languages for different purposes. World of Warcraft is mainly in C++, with probably a bit of assembly for very speed critical code, and Lua for scripting and user interface.

Compare solutions for the N-queens problem on Rosetta Code.

Compiled language

Development:

- write source code
- give source code to compiler (a program on the developer's machine)
- compiler converts source to machine code, thus creating a program
- distribute program

Use:

- ask OS to run program
- machine code is loaded into memory, processor executes instructions directly

(C, C++, D, Rust, Go...)

Interpreted language

("scripting" language)

Development:

- write source code
- distribute source code

Use:

- give source code to interpreter (a program on the user's machine)
- interpreter builds in memory a high-level representation of the source
- interpreter follows this representation and executes commands

Interpreter = simulation of computer (slow). Often embedded in another program (game, application). Fast enough for most purposes.

(Lua, Python, Ruby, JavaScript...)

Bytecode

Development:

- write source code
- give source code to compiler
- compiler converts source to bytecode, creating a "program"
- distribute "program"

Use:

- give "program" to interpreter ("virtual machine", VM)
- interpreter executes bytecode instructions one by one

(Java, C#, ActionScript...)

Bytecode

Advantage:

- separate compilation: can optimize
- portable "program"
- bytecode instructions "almost" machine code: still pretty fast

In practice, most interpreted languages also compile to bytecode, but there is no separate compilation step: program are distributed in source form (Lua, Python, Ruby...)

Choosing a language

The burden of choice

"I want to learn to program. What language should I learn?"

- impossible to answer
- each language adapted to a different purpose
- just learn many!
- easier to go from low-level to high-level (but also harder to start from low-level...)

The serious approach

Learn the C language to:

- understand inner workings of a computer
- learn discipline in programming (because in C you can't do without)
- know what happens behind the scenes of higher level languages (and thus program better with them)
- realize that some seemingly trivial tasks are a pain in the butt

Learn an object-oriented derivative of C for any serious programming job:

- C++ is *the* most popular language to date
- Objective-C, Swift for anything Apple related
- C# for anything Microsoft related
- D is "C++ done right", but is not so popular yet

(when in doubt, all of the above...)

The scripting approach

In parallel, or after, or instead, learn Lua/Ruby/Python:

- basics can be learned quite fast
- seemingly complex tasks can in fact be trivial!
- "real" programming languages
  - actually used in real life
  - available frameworks for desktop applications, video games, web servers...
- less suited for big projects, because they require self imposed discipline

The rest

The *very* serious approach:

- assembly: doing the job of a compiler
- Lisp: old, still used, quite theoretical
- functional languages: another way to think, quite powerful if you can wrap your head around them
  - Haskell
  - Erlang

To get a job but hate yourself:

- Java

(No seriously, don't learn Java.)

There's more!

For web development (ie. dynamic websites):

- frontend: JavaScript
- backend: PHP, Java, Python, Ruby, Erlang, even C/C++
- content: HTML, CSS (not programming languages ☺)

The future:

- Rust
- Go
- D
- to be continued?

Computer architecture

Computer

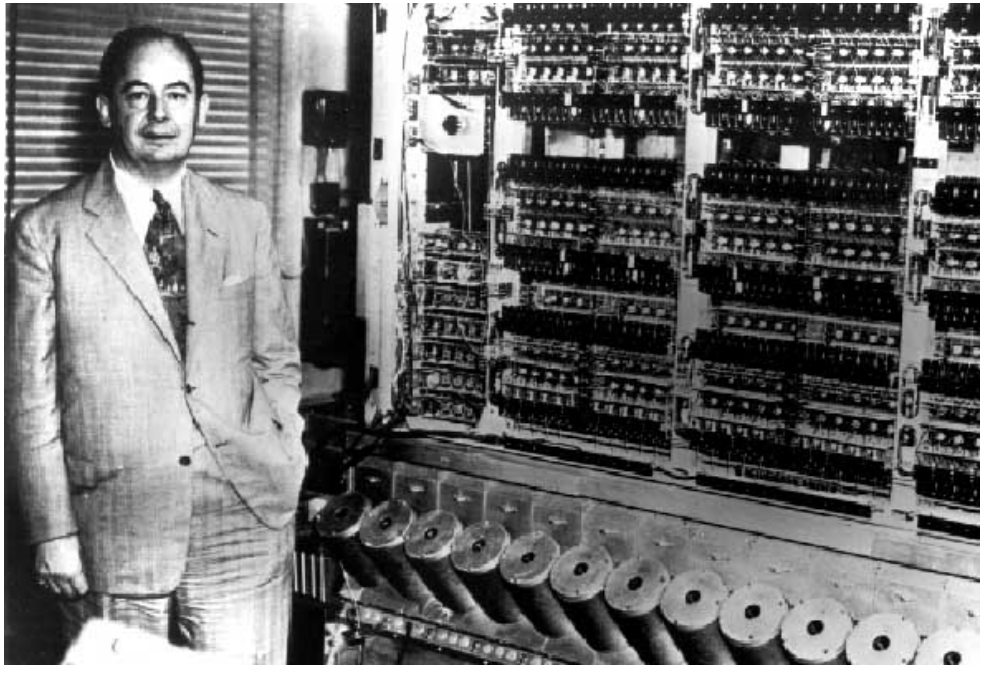
A computer is:

- an electronic device (mechanical, biological?)
- programmable (it runs *programs*, ie. user-defined sequences of instructions)
- processes data (programs read, treat and write information from/to memory and peripherals)

Von Neumann architecture

- processor (CPU: central processing unit)
- memory (RAM: random access memory)
- peripherals

Program: sequence of machine code instructions encoded with binary numbers



Von Neumann architecture (2)

To execute ("run") a program:

- copy entire program somewhere in memory
- point processor at first instruction ("start here")
- processor reads instruction
- processor executes instruction
- processor goes to next instruction
- go on forever

Harvard architecture

- processor
- program memory
- data memory
- peripherals

Advantages:

- can access both memories independently, at the same time: more efficient
- both memories can have different technical characteristics: optimized electronics

In practice, "Modified Harvard architecture":

- common memory for both program and data (allows moving programs around as if they were data)
- different buses and caches for program and data (allows accessing both at the same time)

Operating system

Operating system (itself a program!) takes care of:

- loading programs from hard drive to memory
- cleaning up memory when program is over
- running several programs at the same time
- providing libraries for easier access to hardware





Program and data

Instructions in a program are mostly about:

- manipulating data
  - reading data from memory/peripherals
  - writing data to memory/peripherals
  - performing simple operations on data
    - arithmetics
    - comparison
- jumping to other instructions depending on some data ("if this, then jump"). Otherwise a program would always do the exact same thing, boring.
- that's pretty much it!

Program and data

You can now write Mass Effect 3!

Mozilla Firefox:

- ~15 million LOC
- ~60 MB of machine code
- ~10 million instructions

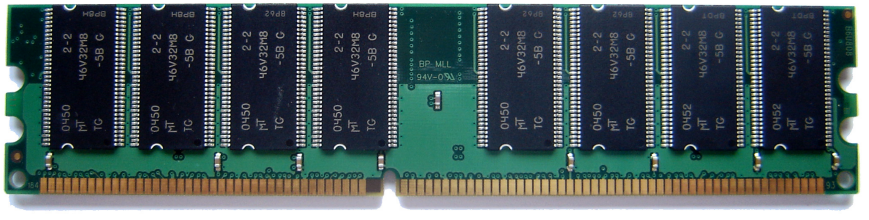
What's data anyway?

Memory is just a big bunch of boxes. Each box:

- can contain a number from 0 to 255. That's a *byte*.
- has an *address*, from 0 to however many bytes of RAM your computer has.

When the processor deals with memory, it looks like this:

- read number from some address
- use it for operation
- write result of operation to some address



What's data anyway?

Anything we want a computer to deal with has to be encoded somehow with bytes.

In practice:

- *bits*: boxes of zeros and ones (because of electronics)
- a byte is 8 bits
- computer always accesses whole bytes

Hence the use of binary numbers (0's and 1's)

Other types

So, integers from 0 to 255 are not enough?

- take two bytes at a time for integers from 0 to 65535
- 4 bytes: 0 to 4294967295
- 8 bytes: 0 to 18446744073709551615

Fortunately, the processor can also read, write and operate directly on integers of these sizes.

- Negative integers? Just interpret the first bit as the sign, and the rest as a positive integer (that's a lie).
- Non integer numbers? Use part of the bits for the integer part, and the rest for the fractional part (that's also a lie).

Other types

- Text: one to one match between characters and numbers (an *encoding*). In ASCII, 'a' is 97, 'W' is 87 and '$' is 36.
- Colors: One number from 0 to 255 for the red component, another for the blue component, another for the green. Three bytes make a pixel.
- Sound? The CD format specifies 2 byte integers for each sample, and 44100 samples per second, each sample representing the intensity of the pressure wave at a given moment.

> It's all numbers!

Notable ladies in computing

Ada Lovelace (1815–1852)

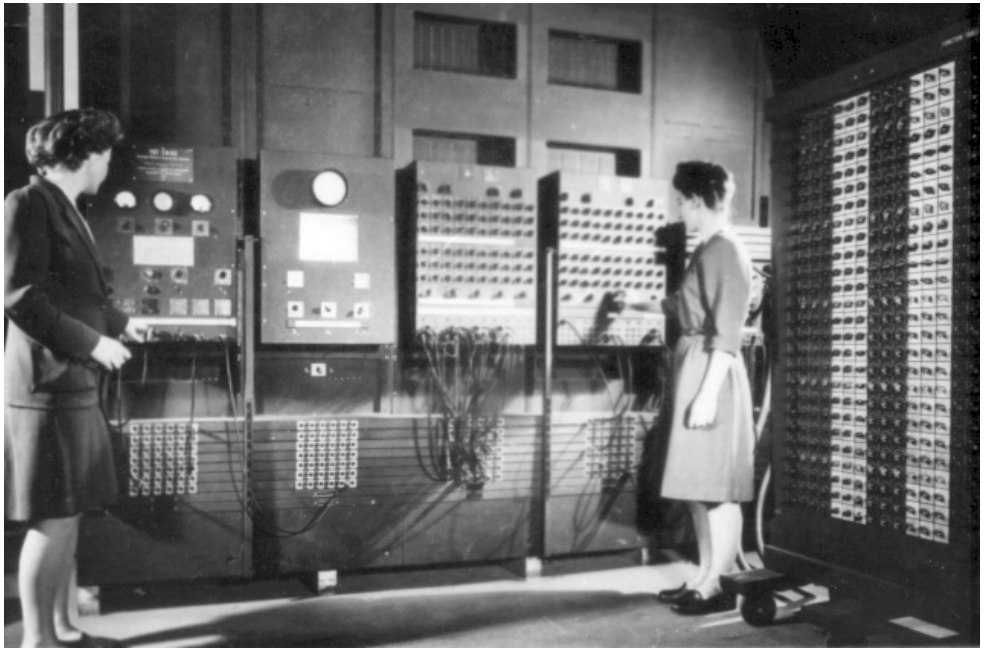
- 1842: wrote the first algorithm to be implemented on Charles Babbage's Analytical Engine (the first mechanical computer)
- the first computer programmer!
- fixed the first computer bug
- programming language Ada named after her

Hedy Lamarr (1913–2000)

- 1942: co-invented frequency-hopping spread spectrum technology to prevent radio jamming during World War 2
- still used for most modern radio technologies: WiFi, Bluetooth, mobile phones, etc.
- also, a famous actress...

ENIAC programmers (1946)

- Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas, Ruth Lichterman
- ENIAC: the first general purpose electronic computer (US Army)



Grace Hopper (1906–1992)

- 1949-1952: invented the first English-like programming languages and their compilers (FLOW-MATIC & MATH-MATIC, for the UNIVAC, the first commercial computer in the US)
- led to COBOL, a language still used today
- also, a rear-admiral in the US Navy...

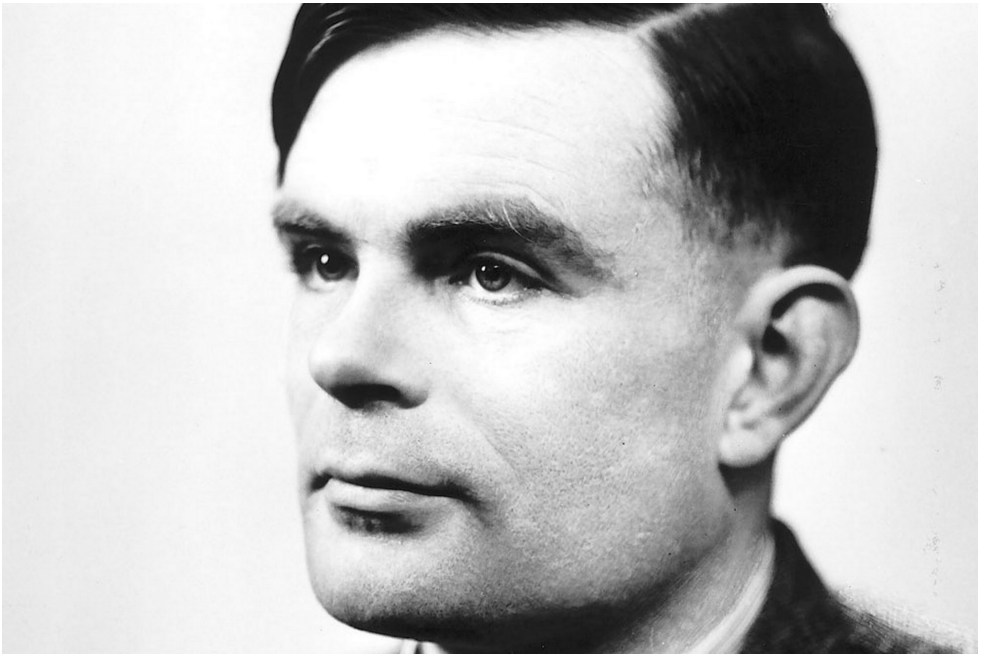
And many more...

See Wikipedia

A few gentlemen too

- Alan Turing (1912-1954)
  - theoretical computer science
  - mathematical formalisation of computers and computability ("Turing machine")
  - philosophical implications of computing
  - artificial intelligence
  - also, pretty much won World War 2 by cracking Enigma...
- Dennis Richie (1941-2011), Ken Thompson (1943-): C, Unix
- Richard Stallman (1953-), Linus Torvalds (1969-): GNU & Linux, free software
- Bjarne Stroustrup (1050-): C++
- Donald Knuth (1938-), Edsger Dijkstra (1930-2002): algorithmics

> I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted. (Alan Turing)



Appendix

Binary numbers

Base 2 positional numeral system. Just like "ordinary" numbers are base 10.

```
25486 = 2 * 10^4  + 5 * 10^3 + 4 * 10^2 + 8 * 10^1 + 6 * 10^0
      = 2 * 10000 + 5 * 1000 + 4 * 100  + 8 * 10   + 6 * 1


10110 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0
      = 1 * 16  + 0 * 8   + 1 * 4   + 1 * 2   + 0 * 1
      = 22
```

Arithmetics work the same:

```
   1101 (13)
+  1001 (9)
-------
= 10110 (22)
```

```
    1101 (13)
x     11 (3)
--------
    1101 (13)
   1101  (26)
--------
= 100111 (39)
```

Binary numbers (2)

- bit: **b**inary dig**it** (0 to 1)
- octet: 8 bits (0 to 255)
- byte: smallest addressable amount of memory (= 1 octet on most modern CPU architectures)
- nybble: half a byte, 4 bits (0 to 15)
- word: the "normal" integer size for the architecture (was 16 bits, now 32 or 64)

SI prefixes confusing. Prefer binary prefixes:

```
k: kilo = 1000          Ki: kibi = 1024   = 1.024 k
M: mega = 1000^2        Mi: mibi = 1024^2 = 1.049 M
G: giga = 1000^3        Gi: gibi = 1024^3 = 1.074 G
T: tera = 1000^4        Ti: tebi = 1024^4 = 1.099 T
```

Hexadecimal numbers

Same positional numeral system, but base 16.

- hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- 1 hex digit = 1 nybble (0 to f)
- 2 hex digits = 1 byte (0 to ff)

Some uses:

- constants in code: `0xdeadbeef` (= 3735928559)
- colors in HTML/CSS: `#c0c0ff = rgb(192, 192, 255)`
- MAC addresses: `01:23:45:67:89:ab`
- ...

Negative numbers

2's complement: invert all bits, add one (ignoring overflow)

```
5 = 0000 0101  ==>  -5 = 1111 1010 + 1 = 1111 1011
```

Why? Does not change the addition operation:

```
  0000 0101 (5)
+ 1111 1011 (-5)
------------
  0000 0000 (0)
```

```
  0001 0101 (21)
+ 1111 0111 (-9)
------------
  0000 1100 (12)
```

```
  0001 1001 (25)
+ 1001 1100 (-100)
------------
  1011 0101 (-75)
```

Floating point numbers

Check Wikipedia if interested.

It's... complicated.