So You Think You Know C?

And Ten More Short Essays on Programming Languages

by Oleksandr Kaleniuk

Published in 2020

This is being published under the Creative Commons Zero license.

I have dedicated this work to the public domain by waiving all of my rights to the work worldwide under copyright law, including all related and neighboring rights, to the extent allowed by law.

You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission.

Table of Contents

Introduction

I've been programming since I was nine. It is my passion, my job, my hobby, and my comfort zone. That's what I do when I'm stressed, angry, or depressed. Also, more often than anything else, it is my reason to be stressed, angry, and depressed in the first place

I never did COBOL. Except for that one time on a dare. But TASM, MASM32, ILAsm, C89, C++17, C#, Prolog, Erlang, Java, JavaScript, Action Script, InstallShield Script, Object Pascal, regular Pascal, Auto Lisp, homemade Lisp, Lisp that is actually a Python, actual Python,—you name it. I tried them all. And a little bit of APL, too.

Why though?

Well, it is a bit like traveling. You get to see the architecture you don't see in your home town, you get to eat food you'd never think about cooking yourself, you get to talk to people with the worldview you never imagine existed. It's fun, it's educational, it's good for you.

But to be completely honest, even more than like traveling, for me it was like running away. I was never happy with professional programming. I was never happy with my own work either.

My first full-time job was to support an eight years old game engine in C++ made by five different programming teams in succession. It had everything: C-ssembler, WinAPI calls, POSIX multithreading, one-liners, thousand-liners, class diagrams that don't fit the screen, and since it was the time of the Modern C++ Design, extensive use of template meta-programming on top.

It was so needlessly fascinating, that every time I could get a chance of doing something on the side, like writing shaders in GLSL or utilities in MASM32 or tech-demos for a pocket console in C, I would take that chance.

I guess this turned into a habit. On my next job, I was still writing in C++ most of the time but I made a documentation system in PHP, wrote a scene editor in

C#, ported our then new and shiny engine to Objective-C and back, and even joined one very controversial project in ActionScript.

Then I tried to switch the stack entirely and started from scratch by joining an Erlang-powered company but somehow still found myself writing Python and JavaScript there.

I found a job in NPP automation that required C++ experience and, guess what, while I started in C++, just in a few months I shifted to Python, C, and Assembly.

On my next job, I finally made peace with C++, and with it, I found my inner peace. Maybe C++ got better or maybe I just got older but I’m not running away anymore. I still enjoy trying other languages such as Julia, Clojure, or Scala but these days it’s more like a series of holiday trips than a journey.

I can’t say that running away from complexity was a smart thing to do. It was definitely not the most productive way to create software. The journey, however, has taught me a few things so it was not a complete waste of time either.

This book is a reflection of these lessons. It consists of small essays each about one particular language and one particular insight.

They can be read in order but they can be read separately. I encourage you to start from the languages you’re the least interested in. I also encourage you to try the actual languages too.

I hope you’ll find this book not too boring, not too trivial, and if nothing else, not too long.

So you think you know C?

A lot of programmers claim they know C. Well, it has the most famous syntax, it has been there for half a century, and it’s not cluttered with obscure features. It’s easy!

I mean, it’s easy to *claim* that you know C. You probably learned it in college or on the go, you probably had some experience with it, you probably think that you know it through and through because there’s not much to know. Well, there is. C is not that simple.

If you think it is—take this test. It only has 5 questions. Every question is basically the same: what the return value would be? And each question has a choice of four answers, of which one and only one is right.

Question 1

```
struct S {
  int i;
  char c;
} s;

int main(void) {
  return sizeof(*(&s));
}
```

A) 4

B) 5

C) 8

D) I don't know.

Question 2

```c
int main(void) {
  char a = 0;
  short int b = 0;
  return sizeof(b) == sizeof(a+b);
}
```

A) 0

B) 1

C) 2

D) I don't know.

Question 3

```c
int main(void) {
  char a = ' ' * 13;
  return a;
}
```

A) 416

B) 160

C) -96

D) I don't know.

Question 4

```
int main(void) {
  int i = 16;
  return (((((i >= i) << i) >> i) <= i));
}
```

A) 0

B) 1

C) 16

D) I don't know.

Question 5

```
int main(void) {
  int i = 0;
  return i++ + ++i;
}
```

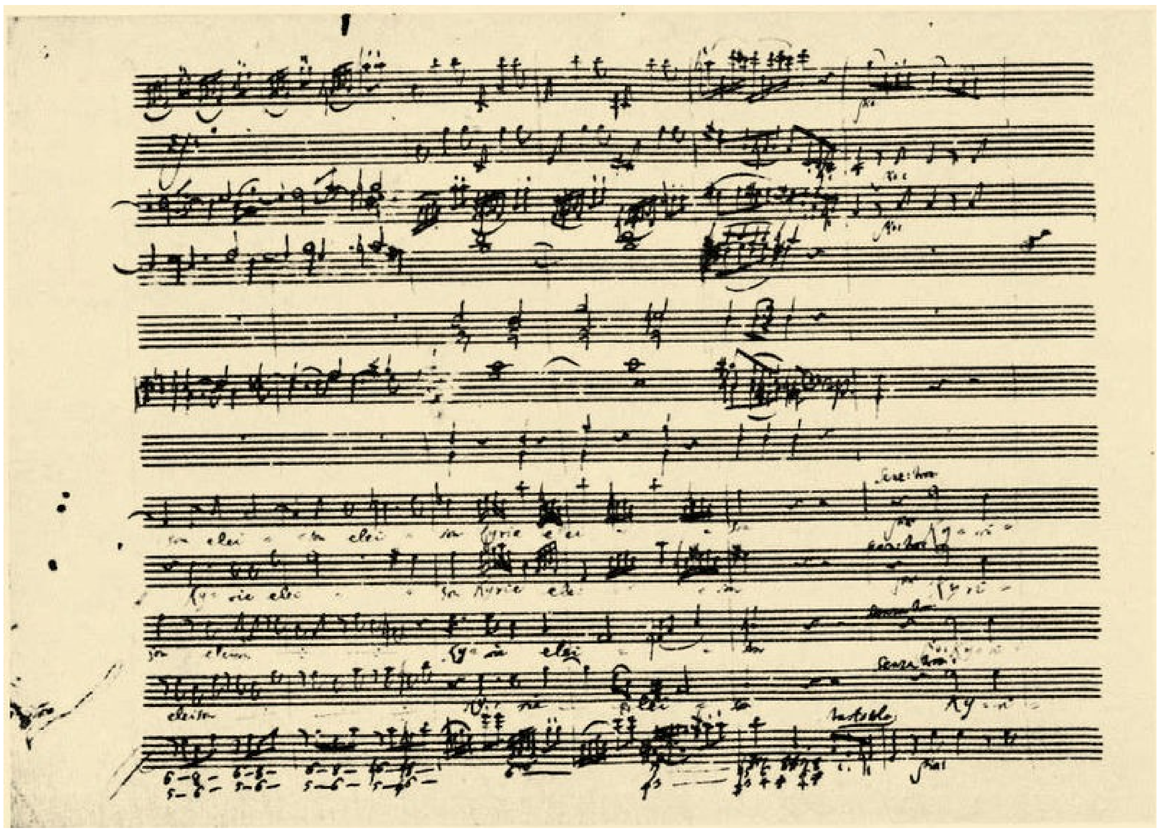
A) 1

B) 2

C) 3

D) I don't know.

That’s it, put down your pens. The answers are under the sheet.



Great Mass in C minor by Wolfgang Amadeus Mozart [Public domain], via Wikimedia Commons.

That's right, Mozart also wrote in C

The right answers are:

	A	B	C	D
1				✔
2				✔
3				✔
4				✔
5				✔

And yes, the right answer to every question[1] is "I don't know"

Let's untangle them one by one.

The first one is about structure padding. C compiler knows that storing unaligned data in RAM may be costly, so it pads your data for you. If you have 5 bytes of data in a structure, it will probably make it 8. Or 16. Or 6. Or whatever it wants. There are extensions like GCC attributes aligned and packed that let you get some control over this process, but they are non-standard. C itself does not define padding attributes, so the right answer is: "I don't know".

The second one is about integer promotion. It's only reasonable that the type of short int and the type of expression with the largest integer being short int would be the same. But "reasonable" doesn't mean "right" for C. There is a rule that every integer expression gets promoted to int. Actually, it's much more complicated than that. Take a peek at the standard, you'll enjoy it.

But even so, we don't compare types, we compare sizes. And the standard only guarantees that the range of short int should not exceed int. Their ranges may be equal, and technically the size of the *short int* may even be greater than the size of *int* because of the padding bits. So the right answer is: "I don't know".

The third one is all about dark corners. Starting from that neither integer overflows, nor char type sign are defined by the standard. The overflows are undefined, and the char sign is implementation-specific. But even more, the size of the *char* type itself is not specified in bits either. Historically, there were platforms where it was 6 bits (remember trigraphs?), and there are platforms today where all five integer types are 32 bits. Even the value of the space character is implementation-defined. Without all these details specified, every speculation about the result is invalid, so the answer is: “I don’t know”.

The fourth one looks tricky, but it’s not that hard in retrospective since you already know that int size is not directly specified in the standard. It can easily be 16 bits, then the very first operation will cause the over-shift and that’s plain undefined behavior. It’s not C fault, on some platforms, it is even undefined in assembly, so the compiler simply can’t give you valid guarantees without eating up a lot of performance.

So once again “I don’t know” is the right answer.

And **the last one** is classic. The order of operand evaluation for + is not specified, and i++ and ++i alter their operand. It might work just like you expect on one platform and might fail easily on the other. Usually, it just evaluates to *2*, so you get used to it until one day it doesn't. That’s the problem with unspecified things. When you meet one, the right answer is always: “I don’t know.”

P. S.

And at this point, I only have to apologize. The test is clearly provocative and may even be a little offensive. I’m sorry if it causes any aggravation.

The thing is, I learned C in roughly 1998, and for the whole 15 years thought that I’m good at it. It was my language of choice in college, and I’ve done some successful projects in C on my first full-time job, and even when I was working in C++, I mostly thought of it as of over-bloated C.

The pivoting moment came in 2013, when I’ve got myself involved with some safety-critical PLC programming. It was a research project in nuclear power plant automation, where absolutely no underspecification was tolerable. I had to learn that, while I did know a lot about C programming, the absolute majority of what I knew was false. And I had to learn it the hard way too.

Eventually, I had to learn to rely on the standard instead of folklore; to trust measurements and not presumptions; to take “things that simply work” skeptically,—I had to learn an engineering attitude. This is what matters the most, not some particular WAT anecdotes.

I only hope this little test would help someone like myself from the past to learn this attitude in some 15 minutes and not 15 years.

APL deserves its renaissance too

This is the Game of Life in APL.

```
Life←{↑1 ω∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

I know, I know. I should have started with the introduction. But doesn't it introduce itself rather well? You can see for yourself that it's ultimately concise, expressive, and utterly alien to all the mainstream computer languages.

In fact, it didn't originate as a computer language at all. It was proposed as a better notation for tensor algebra by Harvard mathematician Kenneth E. Iverson. It was meant to be written by hand on a blackboard to transfer mathematical ideas from one person to another.

But due to its formality, it turned up to be surprisingly good to transfer ideas from people to computers as well. It was made into a computer language in the early 60s. These weird symbols like ↑ or ⌽ were not a problem at all, because every hardware manufacturer had its own keyboard at the time anyway. ASCII wasn't even ratified yet.

Its popularity grew through the following years peaking in the 70s. Fun fact, the very first portable computer by IBM—IBM 5100 "a 50-lb package of interactive personal computing"—came 6 years before the famous IBM PC and with APL on board.

By Marcin Wichary from San Francisco, U.S.A.

([1]Uploaded by Partyzan_XXI) [CC BY 2.0], via Wikimedia Commons

The secret of APL's popularity was simple: learning all the alien symbols is a one-time investment, and expressiveness—the leverage you as a programmer gain—is for life.

However, later on, with the rise of BASIC based personal computing and C powered UNIX platform, APL came off the scene. It is still used in some niches, such as in the financial sector, so there are people who actually make good money using APL up to this day. But it is, of course, as far from the mainstream as it can get.

Still, it's a nice and powerful language. And it's simple too. You might not believe it, but it's one of the simplest languages in existence. I mean, sure, mastering tensor algebra is a bit tricky but once you're there, the language itself is not at all complicated.

Here, let me show you how the Game of Life works[2].

The left arrow is an assignment, and the brackets mark a function's body. So this: **life←{...}** is simply a function definition.

In APL function's arguments are tacit, meaning you don't have to specify a name for every argument, you just know by convention, that the left argument is always α and the right is always ω . But doesn't it mean that APL functions take only two arguments at most? Not really. When you want to call C-like function like this: *foo(x, y, z)*, in APL terms you simply pass a tuple of 3 values as a single argument. It's still one ω . And you can't do *(a, b, c)foo(x, y, z)* in C at all, so APL is actually more powerful than C in this regard.

Let's run our life with some input. We'll use the ρ function to form a 5x5 matrix out of a linear array.

```
1. Form the in variable as a 5x5 matrix

    in ← 5 5 ρ 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 1 1 1 0 0 0 0 0 0

2. Show the in variable

    in

0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
```

In the Game of Life, this figure is called Glider. Running the *life* function on *in* will result in this:

1. Show the *in* variable

```
    in

0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
```

2. Run *life* on *in*

```
    life in

0 0 0 0 0
0 0 0 0 0
0 1 0 1 0
0 0 1 1 0
0 0 1 0 0
```

The Glider moves!

In APL what we would normally call *operators* are *functions* too. Things like these: `+`, `-`, `×`, `,` etc.

The functions are executed from right to left one at the time. There is no precedence, all the functions are equal.

The first function of the *life*'s body would be *enclose*: ⊂ . What it does—it makes our 5x5 matrix input into a scalar containing 5x5 matrix.

1. Show the *in* variable

```
    in

0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
```

2. Enclose the *in* matrix as a scalar

```
    ⊂ in

┌─────────┐
│0 0 0 0 0│
│0 0 1 0 0│
│0 0 0 1 0│
│0 1 1 1 0│
│0 0 0 0 0│
└─────────┘
```

The next thing, as we're going right to left, is a bit trickier. It is *rotate*: `⌽` . It rotates an array at a given index.

```
1. Rotate an array 1 position left

    1 ⌽ 1 2 3 4

2 3 4 1

2. Rotate an array 2 positions left

    2 ⌽ 1 2 3 4

3 4 1 2

3. Rotate an array 1 position right (-1 left)

    ¯1 ⌽ 1 2 3 4

4 1 2 3

4. Rotate an array 0 positions (let it be)

    0 ⌽ 1 2 3 4

1 2 3 4
```

But in our example, it doesn't go by itself. It is itself an argument for an *outer product operator*: `∘.` (in APL, functions that take other functions as arguments are called operators).

And together they do this:

```
1. Enclose the in variable

    ⊂ in

┌─────────┐
│0 0 0 0 0│
│0 0 1 0 0│
│0 0 0 1 0│
│0 1 1 1 0│
│0 0 0 0 0│
└─────────┘

2. Apply outer rotation for -1, 0 and 1

    ¯1 0 1∘.⌽⊂ in

┌─────────┬─────────┬─────────┐
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│
│0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│
│0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
└─────────┴─────────┴─────────┘
```

The next function also goes with an operator. It’s *rotate first*: `⊖`. It works pretty much like *rotate*, but it rotates a nested array around the first level of “nestedness”. Basically, it scrolls matrices.

1. Just show the *in*

```
    in

0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
```

2. Rotate a matrix 1 position up

```
    1 ⊖ in

0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
0 0 0 0 0
```

3. Rotate a matrix 2 positions up

```
    2 ⊖ in

0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
```

4. Rotate a matrix 2 position down (-2 up)

```
    ¯2 ⊖ in

0 1 1 1 0
0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
```

5. Rotate a matrix 1 position down (-1 up)

```
    ¯1 ⊖ in

0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
```

6. Rotate a matrix 0 positions (let it be)

```
    0 ⊖ in

0 0 0 0 0
0 0 1 0 0
0 0 0 1 0
0 1 1 1 0
0 0 0 0 0
```

With the outer product operator and our previous result it goes like this:

1. Enclose the *in* variable

```
      ⊂ in
 ┌─────────┐
 │0 0 0 0 0│
 │0 0 1 0 0│
 │0 0 0 1 0│
 │0 1 1 1 0│
 │0 0 0 0 0│
 └─────────┘
```

2. Apply outer *rotate* for -1, 0 and 1

```
      ¯1 0 1∘.⌽⊂ in
 ┌─────────┬─────────┬─────────┐
 │0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
 │0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│
 │0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│
 │0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│
 │0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
 └─────────┴─────────┴─────────┘
```

3. Apply outer *rotate first* for -1, 0 and 1

```
    ¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

┌─────────┬─────────┬─────────┐
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│
│0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│
│0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│
├─────────┼─────────┼─────────┤
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│
│0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│
│0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
├─────────┼─────────┼─────────┤
│0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│
│0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│
│0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│
└─────────┴─────────┴─────────┘
```

The next function in called *ravel*: `,` and it looks like a comma. What it does, it makes a nested array 1-dimensional.

```
1. Form the x variable as a 2x3 matrix

    x ← 2 3 ρ 1 2 3 4 5 6

2. Show the x variable

    x

1 2 3
4 5 6

3. Use ravel to make a matrix back into an array

    , x

1 2 3 4 5 6
```

It doesn't ravel scalars, so being applied to our 3x3 matrix of enclosed matrices, it would make a linear array of 9 enclosed matrices.

```
    ,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

┌─────────┬─────────┬─────────┬───...───┬─────────┐
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│         │0 1 0 0 0│
│0 0 0 0 0│0 0 0 0 0│0 0 0 0 0│         │0 0 1 0 0│
│0 0 0 1 0│0 0 1 0 0│0 1 0 0 0│   ...   │1 1 1 0 0│
│0 0 0 0 1│0 0 0 1 0│0 0 1 0 0│         │0 0 0 0 0│
│0 0 1 1 1│0 1 1 1 0│1 1 1 0 0│         │0 0 0 0 0│
└─────────┴─────────┴─────────┴───...───┴─────────┘
```

The next piece of code is an operator and function pair. Operator *reduce*: `/`, and a function *plus*: `+`. As you might guess, it reduces, as in map-reduce, a summation of all the matrices in a linear array.

```
      +/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

 ┌─────────┐
 │0 1 1 1 0│
 │0 1 2 2 1│
 │1 3 5 4 2│
 │1 2 4 3 2│
 │1 2 3 2 1│
 └─────────┘
```

Operator *compare*: `=` produces matrices of 0 and 1 based on whether every element in the right argument equals a corresponding element in the left argument. In our case we would use it to filter out 3s and 4s:

1. Sum all the rotated matrices

```
    +/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

┌─────────┐
│0 1 1 1 0│
│0 1 2 2 1│
│1 3 5 4 2│
│1 2 4 3 2│
│1 2 3 2 1│
└─────────┘
```

2. Compare with 4. All the 4s are now 1

```
    4 = +/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

┌─────────┐
│0 0 0 0 0│
│0 0 0 0 0│
│0 0 0 1 0│
│0 0 1 0 0│
│0 0 0 0 0│
└─────────┘
```

3. Compare with 3 and 4

```
    3 4 = +/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

┌─────────┬─────────┐
│0 0 0 0 0│0 0 0 0 0│
│0 0 0 0 0│0 0 0 0 0│
│0 1 0 0 0│0 0 0 1 0│
│0 0 0 1 0│0 0 1 0 0│
│0 0 1 0 0│0 0 0 0 0│
└─────────┴─────────┘
```

Then there is the logical part. Functions *or*: `∨` and *and*: `∧` used with the *inner product* `.` operator. They form an operator that makes an *and* on every element pair and then an *or* on a result.

```
1. And function

    1 0 1 0 ∧ 1 1 0 0

1 0 0 0

2. Or function

    1 0 1 0 ∨ 1 1 0 0

1 1 1 0

3. Inner product operator of or-ed ands

    1 0 1 0 ∨.∧ 1 1 0 0

1
```

Remember, we didn't count the neighbors exactly, we counted them along with the value in each cell. This was the 0 0 rotation. Now we have to build our logic around this.

- If the sum is 3 and the cell had 0, then it has 3 neighbors—it lives.
- If the sum is 3 and the cell had 1, then it has 2 neighbors—it still lives.
- If the sum is 4 and the cell had 0, then it has 4 neighbors—it doesn't repopulate.
- If the sum is 4 and the cell had 1, then it has 3 neighbors—it lives.

That's why when we want our *and* to work with 3s unconditionally, we supply it with identity matrix 1. And when we want to and 4s with the original input, we supply it with *in*. Then we only have to *or* the results so any condition produces a living cell.

That's how **1 in ∨.∧ 3 4...** works.

```
      1 in ∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

 ┌─────────┐           ⍝   Identity      Original in
 │0 0 0 0 0│           ⍝     1 1 1 1 1     0 0 0 0 0
 │0 0 0 0 0│           ⍝     1 1 1 1 1     0 0 1 0 0
 │0 1 0 1 0│           ⍝     1 1 1 1 1     0 0 0 1 0
 │0 0 1 1 0│           ⍝     1 1 1 1 1     0 1 1 1 0
 │0 0 1 0 0│           ⍝     1 1 1 1 1     0 0 0 0 0
 └─────────┘
```

The last function *mix* ↑ here simply removes the enclosure.

```
1. Enclosed

      1 in ∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

 ┌─────────┐
 │0 0 0 0 0│
 │0 0 0 0 0│
 │0 1 0 1 0│
 │0 0 1 1 0│
 │0 0 1 0 0│
 └─────────┘

2. Mixed

      ↑1 in ∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂in

0 0 0 0 0
0 0 0 0 0
0 1 0 1 0
0 0 1 1 0
0 0 1 0 0
```

Not so alien now, is it?

But why does it deserve its renaissance after all?

The last decade was a renaissance for the Lisp being reborn in Clojure. Also, Erlang, being a niche language for telecom, decided to waltz into the mainstream with a plethora of web frameworks. Even Haskell not only gained popularity on its own but deeply influenced F# and Scala. APL however, despite its tremendous power, didn't get too much acclaim.

My theory is, since it was designed to be written by hand, it didn't work out very well with an ASCII. Didn't survive the standardization. Of course, APL has its ASCII friendly descendants inherited its expressiveness but frankly, they are all ugly far beyond the possibility of public success. They simply lack the style.

Roughly 86% of all the fun you get from APL programming comes from the mysterious symbols and the magic behind them.

So it is not that APL is alien to computers, it’s just the computers were alien to APL for quite a while.

But now, with the development of touch interfaces and optical character recognition, it might just get its second chance. Since on a tablet or on a phone you do have to use a virtual keyboard anyway, why not choose APL for its tablet-friendly concision? Or you could even draw APL symbols with your fingers. Not being constrained with the keyboard, you could name your own functions with hand-written symbols as well. You could develop your own truly free-form language just like Iverson did!

Since the ratification of ASCII, our languages were held hostage by the standard keyboard. But since the keyboard itself is not a thing for the majority of devices, shouldn’t it lead to the reinvention of more concise and expressive ways to write your code?

Going beyond the idiomatic Python

People don’t speak entirely in idioms unless they are totally off their rockers. Overusing idioms makes you seem more than self-confident, full of air, and frankly not playing with a full deck. It is fair to middling to spice your language with idioms a little bit, but build the whole speech entirely out of them is beside the point.

What I’m trying to say, stuffing your text with idioms you happen to read somewhere doesn’t automatically make it better. And it doesn’t work for your code either.

There is a book called “Writing Idiomatic Python” written by Jeff Knupp. It is a decent collection of Python idioms, mixed together with some of the less known language features and best practices. While being rather helpful as a dictionary or a bestiary, it promotes the dangerous fallacy that writing code idiomatically automatically makes it better.

You might think I’m exaggerating so here is a direct quote:

> *Each idiom that includes a sample shows both the idiomatic way of implementing the code as well as the “harmful” way. In many cases, the code listed as “harmful” is not harmful in the sense that writing code in that manner will cause problems. Rather, it is simply an example of how one might write the same code non-idiomatically. You may use the “harmful” examples as templates to search for in your own code. When you find code like that, replace it with the idiomatic version.*

— Idiom, sir? — Idiom!

Even in that very book, however, there are examples that simply don’t work all that well. Like this one:

```
def contains_zero(iterable):
    # 0 is "Falsy," so this works
    return not all(iterable)
```

This is a trivial function, but due to the unwanted idiom, it requires translation. *If not all elements are iterable then they contain zero?* That’s just nonsense! Using a standard language facility makes the whole thing trivial again.

```
def contains_zero(container):
    return 0 in container
```

The container contains zero if there is a zero in the container. This is so trivial; it doesn’t even deserve to be a function, not to mention having a dedicated comment.

There is another example:

```python
def should_raise_shields():
    # "We only raise Shields when one or more giant robots attack,
    # so I can just return that value..."
    return number_of_evil_robots_attacking()
```

Raise shields when one or more giant robots attack. Ok, that makes sense. But if the logic is clear, why not wire it to the code directly?

```python
def should_raise_shields():
    return number_of_evil_robots_attacking() >= 1
```

It’s 3 more symbols, but they make the function absolutely transparent for a reader. There is no need for a comment anymore.

It’s tempting to think that following some rules will automatically make you a better programmer. But it doesn’t work this way. It’s not about the rules, it’s about when to follow them and when not.

Here, let me show you.

Rule 1. Use long descriptive names when necessary

Before:

```
def inv(a):
    return adj(a) / det(a)
```

After:

```
def inverse_of(matrix):
    return adjugate_of(matrix) /
determinant_of(matrix)
```

Rule 2. Use short mnemonic names when possible

Before:

```
linsolve([matrix_value_b + resuting_point_x0 -
resuting_point_x1*matrix_value_h - resuting_point_x1,
          matrix_value_e + resuting_point_y0 -
resuting_point_y1*matrix_value_h - resuting_point_y1,
          matrix_value_a + resuting_point_x0 -
resuting_point_x2*matrix_value_g - resuting_point_x2,
          matrix_value_d + resuting_point_y0 -
resuting_point_y2*matrix_value_g - resuting_point_y2],
  (matrix_value_a, matrix_value_b, matrix_value_d,
matrix_value_e))
```

After:

```
linsolve([b + x0 - x1*h - x1,
          e + y0 - y1*h - y1,
          a + x0 - x2*g - x2,
          d + y0 - y2*g - y2],
         (a, b, d, e))
```

Rule 3. Prefer function names to comments for clarity

Before:

```
def are_all_numbers_in(list_of_everything):
    for element in list_of_everything:
        try:
            # throws ValueError
            # if element isn’t a number
            float(element)
        except ValueError:
            return False
    return True
```

After:

```
def fails_on_cast_to_float(s):
    try:
        float(s)
        return False
    except ValueError:
        return True

def are_all_numbers_in(list_of_everything):
    for element in list_of_everything:
        if fails_on_cast_to_float(element):
            return False
    return True
```

Rule 4. Don’t clarify what is already clear enough

Before:

```
def hash_of_file(name):
    with open(name, 'r') as text:
        return str(hash(text.read()))

def file_is_xml(file_name):
    return file_name.endswith('.xml')

def name_belongs_to_file(file_name):
    return os.path.isfile(file_name)

def name_belongs_to_xml(name):
    return name_belongs_to_file(name) \
           and file_is_xml(name)

def print_name_and_hash(name, xml_hash):
    print name + ': ' + hash_of_file(file_name)

def traverse_current_directory(do_for_each):
    for file_name in os.listdir('.'):
        do_for_each(file_name)

def print_or_not_hash_for(file_name):
    if name_belongs_to_xml( file_name ):
        xml_hash = hash_of_file(file_name)
            print_name_and_hash(file_name, xml_hash)

traverse_current_directory(print_or_not_hash_for)
```

After:

```
for file_name in os.listdir('.'):
    if os.path.isfile( file_name ) \
    and file_name.endswith('.xml'):
        with open(file_name, 'r') as xml:
            hash_of_xml = str(hash(xml.read()))
        print file_name + ': ' + hash_of_xml
```

Rule 5. Use list comprehension to transform lists

Before:

```
def matrix_of_floats(matrix_of_anything):
    n = len(matrix_of_anything)
    n_i = len(matrix_of_anything[0])
    new_matrix_of_floats = []
    for i in xrange(0, n):
        row = []
        for j in xrange(0, n_i):
            row.append(
                float(matrix_of_anything[i][j]))
        new_matrix_of_floats.append(row)
    return new_matrix_of_floats
```

After:

```
def matrix_of_floats(matrix_of_anything):
    return [[float(a_ij) for a_ij in a_i]
            for a_i in matrix_of_anything]
```

Rule 6. Just because you can do anything with list comprehensions, doesn’t mean you should

Before:

```
def contains_duplicate(array):
    return sum([sum([1 if a_i-rot_ij else 0
                for a_i, rot_ij in zip(array, rot_i)])
                for rot_i in [array[i:] + array[:i]
                for i in range(1, len(array))]]
               ) != len(array) * (len(array) - 1)
```

After:

```
def contains_duplicate(array):
    for i in range(len(array)):
        for j in range(i):
            if array[i] == array[j]:
                return True
    return False
```

Conclusion

I hope you see the pattern. Each even rule seemingly contradicts the odd one. But there is no actual contradiction because they all depend on the context. Every rule is so pointless without the context, it can’t even contradict another rule properly.

Unfortunately, this means that you can’t learn to write good code *only* by following the rules. It’s tempting to think so, but it just doesn’t work this way. It is, of course, good to know idioms and best practices, but it never ends there. You have to go beyond the idioms. Beyond the rules.

As far as I’m concerned, there is only one sure way to improve your coding skills, but it’s so straightforward and unappealing, no one would write a book on it. No one would promote it at a conference. There are, however, seldom blog posts on the topic, but they largely go unnoticed.

So here it is. The one truly working way.

To learn to write good code you have to write a shit-metric-ton of bad code.

And that’s it. Do wrong. Make mistakes. Learn from them. You are a programmer, not a surgeon, or a race car driver. You can afford to practice on your own mistakes and not to kill anyone!

This is a privilege, enjoy it.

Why Erlang is the only true computer language

Imagine there is something wrong with the water pressure in your house. You have to call the plumber service. They say that there is a guy, who is honest, and capable, and has a Ph.D. in plumbing, but… He doesn’t speak English well. He came from Tajikistan and only speaks fluent Tajik. But it just so happens that you studied Tajik in college, so you say that they send the guy immediately.

When he arrives, you describe the problem (in Tajik), and he gets to work. Soon enough he finds the problem and tries to report it to you, apparently waiting for a reaction. But for some reason he doesn’t use Tajik to report it, he uses English. Which… He doesn’t really know. Even if you tell him explicitly (in Tajik) that you don’t understand, he will not switch to Tajik; he will only repeat the same phrase louder.

Which doesn’t help to build a dialog at all.

Nobody knows why he consistently chooses to reply in the language he doesn’t speak well instead of his own. That you do, by the way. It is absurdly irritating, but that’s just how things work. It’s how they have always been. Whatever he’s meaning to say, makes perfect sense in his own tongue. But what he speaks aloud is basically a bad English mixed with professional jargonisms and Tajik obscenities.

Which feels somehow like this:

```
In function ‘int main()’: 19:16: error: cannot bind
‘std::ostream {aka std::basic_ostream<char>}’ lvalue
to ‘std::basic_ostream<char>&&’ In file included from
/usr/include/c++/4.9/iostream: 39:0, from 2:
/usr/include/c++/4.9/ostream:602:5: note:
initializing argument 1 of
‘std::basic_ostream<_CharT, _Traits>& std::operator<<
(std::basic_ostream<_CharT, _Traits>&&, const _Tp&)
[with _CharT = char; _Traits = std::char_traits<char>;
_Tp = Element]’ operator<<basic_ostream<_CharT,
_Traits>&& __os,
const _Tp& __x)
```

Ok, it isn't about plumbing anymore. It never was about plumbing. It's about how we communicate with the machine. A programming language is something we use to tell it what to do, but the machine tries to speak back in English when something gets wrong. Computers don't speak English well, although they are good at processing formal languages. And presumably, the one who writes in a formal language should be able to read it back without a problem. Wouldn't it be better, if the compiler or interpreter would respond in the same language it gets its commands in?

So far the only languages I know that report errors in itself are Erlang, Elixir which is a successor of Erlang, and Ocaml that abandons this feature in favor of English. If you know more, please let me know.

Here's a simple example from "Programming Erlang":

```
1> Point = {point, 10, 45}.
2> {point, C, C} = Point.
=ERROR REPORT==== 28-Oct-2006::17:17:00 ===
Error in process <0.32.0> with exit value:
{{badmatch,{point,10,45}},[{erl_eval,expr,3}]}
```

The string in bold is the error message, the rest is just a preamble from the interpreter. Indeed, it isn't entirely convenient for the casual reader. You have to know Erlang to read its errors. But since you still need to know Erlang to write the code that causes them, it's fine.

The alternatives are much worse. With informal and non-standardized error messaging you have to know not only the programming language itself but its error messaging Pidgin as well. Every C++ practitioner can confirm that reading STL messages is an art by itself. It takes knowledge, patience, determination, and occasionally a crystal ball to build a firm understanding of a reported problem.

For instance, the error message from above reads as "sorry, I can't print out the variable for you, because its type doesn't have a << operator."

Normally, the language is something you can build a dialog in. Programming languages are an unlucky exemption. Historically, it was more important to tell a computer what to do rather than the opposite. When you write a small single-threaded program you can hope that there will be no errors whatsoever so you don’t really need a reporting at all. Error reporting is viewed as an auxiliary feature.

Erlang, however, aims at building highly complex distributed systems. Errors are not only most likely to happen, they are inherent to the nature of the domain. Everything that could go wrong eventually will. And you better have a decent reporting language for that. And since Erlang is a decent language on its own, well, it all just makes sense.

In Erlang, you can have a two-way dialog between you and the machine. The very fact that *you can have a conversation with a computer*, makes Erlang not only a programming language but a true computer language.

The invisible Prolog in C++

It is well known that C++ is a multi-paradigm programming language. It combines procedural, functional, object-oriented, and generic programming features in an arsenal of things to shoot your foot with.

A little less known that it also employs the logic programming paradigm. It doesn't expose its features directly, yet understanding the paradigm while not enriching your arsenal per se makes you much less prone to metaphorical self-injuries.

What's logic programming?

Logic programming is about making the computer deduce facts for you. You write down the things you know, write down the rules that hold true for these things, and then you ask the question. Like, “who killed John F. Kennedy”?

Of course, the computer can only work with the facts you provide. It doesn't do surveillance or interrogation for you. It doesn't have the intuition. But it does the deduction fast and robust, that's all we can expect from the machine.

The most popular logic programming language now is Prolog. It was invented in France, and its name stands for “programmation en logique”, which is in itself quite logical.

Let's take a quick tour of Prolog to see what we can learn from it.

In Prolog you have **terms** to store data. Terms are:

- **atom—basically any word or even a full sentence. Like: *x*, *alice*, or *‘year 2016’*.**
- **number**—floating point or integer.
- **compound term**—complex data type constructed of atoms and numbers. This includes lists and strings.

You build relations between data with rules and facts. When you want to say that “*Alice* ***likes*** *Bob*”, *Alice* and *Bob* are the data and ***likes*** is the relation. In Prolog, you can write it down like this.

```
likes(alice, bob).
```

Such relations are called **facts** in Prolog. There are also **rules** which are conditional facts. Let’s say Alice likes someone who is kind, and intelligent and writes in C++. The rule for that would be:

```
likes(alice, Person) :-
  kind(Person),
  intelligent(Person),
  writes(Person, cpp).
```

The *Person* is a Prolog **variable**. Syntactically, variables always begin with a capital letter. Semantically, they can denote any term that fits the conditions.

Logic programming is all about deduction. You have a set of terms, known facts, and rules. Then you want to know something you didn’t know before. For instance, if we're going to know if Alice likes Bob according to her rule above, we have to introduce Bob with a set of facts and then ask Prolog like this:

```
?- likes(alice, bob).
```

Prolog programming is declarative. This means that we only have to give it rules and facts, but not the way the facts and rules should be checked. Prolog finds the way for us.

So, given the following set of facts, does Alice like Bob or not?

```prolog
kind(bob).
kind(george).
kind(steven).
intelligent(bob).
intelligent(steven).
writes(bob, cpp).
writes(bob, assembly).
writes(george, cpp).
writes(steven, prolog).

likes(alice, Person) :-
  kind(Person),
  intelligent(Person),
  writes(Person, cpp).

?- likes(alice, bob).
```

Yes, she does. Bob is kind, intelligent, and writes in C++. According to our facts and rules, Alice likes him.

Analogies in C++

C++ doesn't have logic deduction as a language feature. But it has something similar. It has type deduction which is very close conceptually. Now let's translate our Prolog program into C++.

Classes will be our atoms.

Polymorphic functions will be our facts.

And a template function will be our rule.

```
// people
class Alice{};
class Bob{};
class George{};
class Steven{};

// languages
class Cpp{};
class Prolog{};
class Assembly{};

// facts
void kind(Bob);
void kind(George);
void kind(Steven);
void intelligent(Bob);
void intelligent(Steven);
void writes(Bob, Cpp);
void writes(Bob, Assembly);
void writes(George, Cpp);
void writes(Steven, Prolog);
```

```
// the rule
template <typename Person> void likes(Alice, Person
person)
{
    kind(person);
    intelligent(person);
    writes(person, Cpp());
}

// check the rule for Bob
int main()
{
    likes(Alice(), Bob());
}
```

Type deduction is not entirely the same as logic programming, but in many regards, it works the same. The program compiles only if the compiler deduces all the types correctly. The very fact of compilation is the answer to our question.

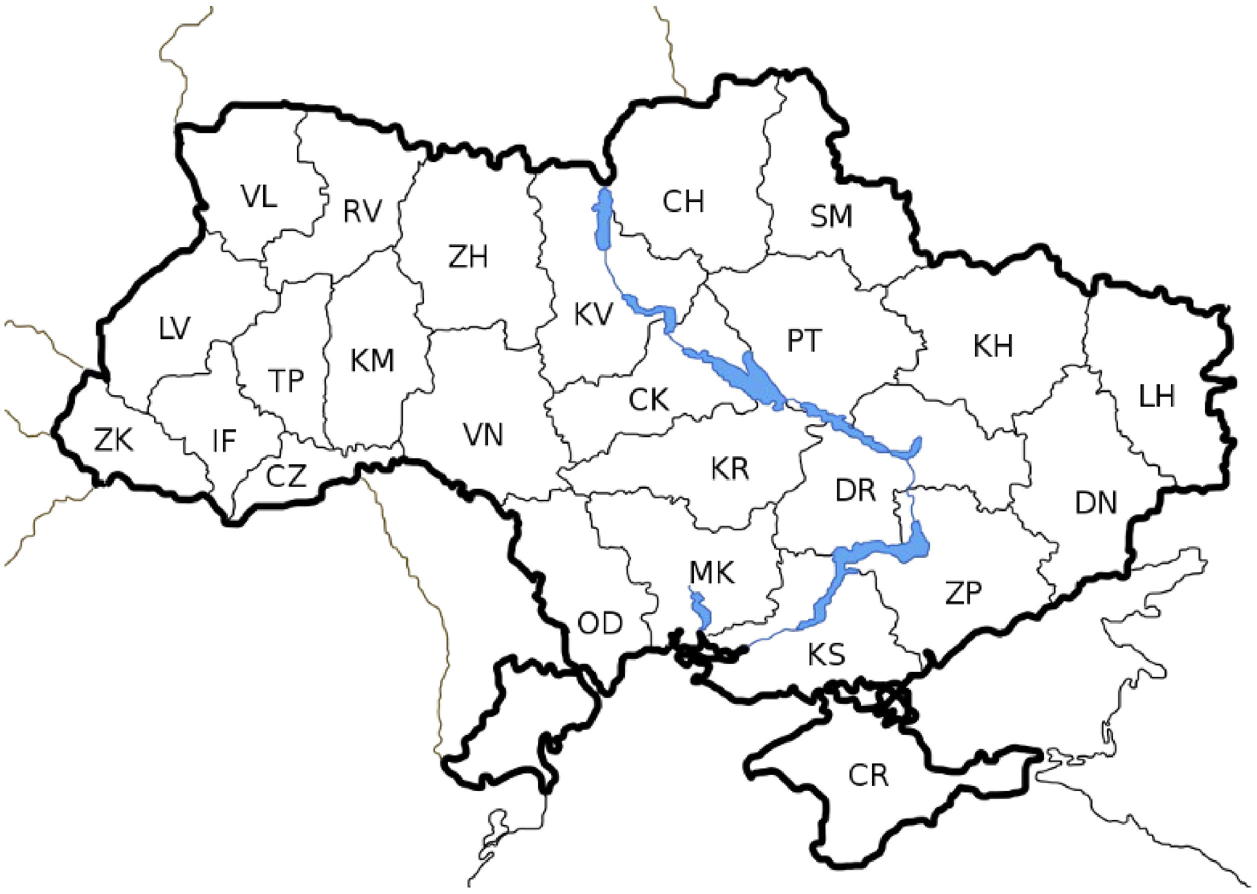
So does Alice like Bob in C++?

Yes, she still does. The program compiles only if there is a compilable *likes* function for *Alice* and *Bob*. And our only defined *likes* is only compilable for *Bob* if there are compilable *kind*, *intelligent*, and *writes Cpp* functions for him. And there are.

Logic programming v.s. type deduction

While being similar, type deduction differs from logic programming in one crucial way. I guess it would be best to illustrate it with the example.

I stole this idea from Bernardo Pires. If you got interested in Prolog and logic programming in general, please read his article. He uses Prolog to color map of Germany in four colors. We will try to do the same with C++ and the map of Ukraine.



By Albedo-ukr [CC BY-SA 2.5], via Wikimedia Commons

At first, we would have to define colors.

```
// colors
class Yellow{};
class Blue{};
class White{};
class Green{};
void color(Yellow);
void color(Blue);
void color(White);
void color(Green);
```

We need to generalize them to use in our rules, and that's one possible way to do that.

```
// AnyColor object can be Yellow, Blue, White or Green
class AnyColor : public Yellow, Blue, White, Green {};
```

Unlike C++, Prolog has an operator to declare data inequality. So when Bernardo wants to declare a rule stating that all the neighboring regions should have different colors, he writes this:

```
neighbor(StateAColor, StateBColor) :-
color(StateAColor),
    color(StateBColor),
    StateAColor \= StateBColor.
```

We can do the same in modern C++, but it gets needlessly tricky, so we'll define inequality as a simple set of facts instead.

```
// color inequality (instead of \= orerator)
void different(Yellow, Blue);
void different(Yellow, White);
void different(Yellow, Green);
void different(Blue, Yellow);
void different(Blue, White);
void different(Blue, Green);
void different(White, Yellow);
void different(White, Blue);
void different(White, Green);
void different(Green, Yellow);
void different(Green, Blue);
void different(Green, White);
```

Next, we want every two adjacent regions to have different colors. Here's a rule for that.

```
// neighborhood rule
template <typename Region1Color, typename
Region2Color>
void neighbor(Region1Color, Region2Color)
{
    color(Region1Color());
    color(Region2Color());
    different(Region1Color(), Region2Color());
}
```

Now we have to program the map of Ukraine as pairs of adjacent regions.

```
// map: neighborhood of regions
template <typename ZK, typename LV, typename IF,
          typename VL, typename CZ, typename TP,
          typename RV, typename KM, typename ZH,
          typename VN, typename OD, typename KV,
          typename CK, typename CH, typename MK,
          typename KR, typename PT, typename KS,
          typename SM, typename DR, typename CR,
          typename ZP, typename KH, typename DN,
          typename LH>
void ukraine(ZK zk, LV lv, IF iv, VL vl, CZ cz, TP tp,
             RV rv, KM km, ZH zh, VN vn, OD od, KV kv,
             CK ck, CH ch, MK mk, KR kr, PT pt, KS ks,
             SM sm, DR dr, CR cr, ZP zp, KH kh, DN dn,
             LH lh)
{
  neighbor(zk, lv);neighbor(zk, iv);neighbor(lv, vl);
  neighbor(lv, rv);neighbor(lv, tp);neighbor(lv, iv);
  neighbor(iv, tp);neighbor(iv, cz);neighbor(vl, rv);
  neighbor(tp, rv);neighbor(tp, km);neighbor(tp, cz);
  neighbor(cz, km);neighbor(cz, vn);neighbor(rv, km);
  neighbor(rv, zh);neighbor(km, zh);neighbor(km, vn);
  neighbor(zh, kv);neighbor(zh, vn);neighbor(vn, kv);
  neighbor(vn, ck);neighbor(vn, kr);neighbor(vn, od);
  neighbor(od, kr);neighbor(od, mk);neighbor(kv, ch);
  neighbor(kv, pt);neighbor(kv, ck);neighbor(ck, pt);
  neighbor(ck, kr);neighbor(ch, sm);neighbor(ch, pt);
  neighbor(mk, kr);neighbor(mk, dr);neighbor(mk, ks);
  neighbor(kr, pt);neighbor(kr, dr);neighbor(pt, sm);
  neighbor(pt, kh);neighbor(pt, dr);neighbor(sm, kh);
  neighbor(ks, cr);neighbor(ks, zp);neighbor(dr, kh);
  neighbor(dr, dn);neighbor(dr, zp);neighbor(zp, dn);
  neighbor(kh, lh);neighbor(kh, dn);neighbor(dn, lh);
}
```

And finally, we write the function that starts type deduction for every region.

```
// try to color map of Ukraine
int main()
{
  ukraine(
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor(),AnyColor(),AnyColor(),AnyColor(),
    AnyColor());
}
```

Given that there are no typos and we wrote down all our rules and facts correctly, will this program compile or not?

No, it will not. Although there are a lot of possible colorings, the compiler wouldn't find them. The crucial difference between type deduction and logic programming is that *type deduction is unambiguous.*

You can color a map in many different ways, but the compiler has to produce one and only one program.

Conclusion

When you write in C++ you actually write in two languages at once[3].

First is C++, and the second one is invisible Prolog.

If written properly, the second program is ultimately helpful. If you build your type relations right, every compilation will reassure you that your expectations about the entity relations are also correct. Type deduction will work just as the logic deduction. Pragmatically, this means fewer bugs and fewer surprises in general.

However, if being neglected, it turns your code into an untangleable mess of incomprehencibles really-really fast. Every new rule not only adds but multiplies the complexity, so it tends to grow in geometric progression.

And that's why acknowledging the invisible language is even more important than mastering the visible one.

One reason you probably shouldn’t bet your whole career on JavaScript

I fell in love with JavaScript long before it was cool. I’m not sure about the exact date and time, but definitely before StackOverflow and probably after GMail announced its beta. Back in the day, it was used mostly to clutter innocent web sites with bells and whistles nobody wanted. Like these January waltzing snowflakes reminding you that Christmas is over and so is bourbon and you have no friends.

A roommate of mine had friends, and he introduced me to the guy who was deep in this emerging front-end thing. And this guy told me the two biggest secrets of JavaScript: it’s not Java, and it’s not script. Well, everybody knows this *now*, but at that time knowing that you have a full-power multi-paradigm language in basically every browser in the world was like urban legend coming to life. It was like learning that your favorite toothpaste is actually made of cocaine, you just don’t rub it into your dents hard enough.

I got hooked. And I wrote pretty bizarre things in JavaScript. Like a multivariate function optimizer, that could infer variable names from a string expression, and then find their values in a local optimum. All in less than 100 lines of code. Not because I really had to, though, but because I could. The language combining dynamic typing with functional capabilities, JavaScript was arguably the most versatile, the most “easy to do weird things” language between the mainstream ones.

Now, of course, everything is made of cocaine. Every mainstream language is multi-paradigm now, every one has functional programming; and dynamic typed ones have type annotations, and static typed ones have variable types instead. And this brings us to an unpleasant conclusion. JavaScript is just not that exciting as a language anymore.

Good ideas are not a privilege now, but a commodity. They transfer freely from one standardization committee to another making all the mainstream languages more and more alike. If I start jogging tomorrow and change my drinking preferences to carrot juice, I could live up to that time when all the mainstream programming languages will become the same.

JavaScript was holding its dominance in front-end for too long. Basically, all its glory comes not from the ingenious design, but from the simple thing that it became de-facto standard on the wave of rising bandwidth and processor power. Now, as everybody gets more focused on performance and power economy, the whole thing is being redesigned.

Come to think about, the way things work now *is* rather stupid. I mean, in order to have depressive snowflakes on your screen, you have to get a JavaScript source from a server, then run a lexical analysis on it, syntax analysis and semantic analysis, then a bunch of machine-dependent optimizations; and only then the essential part — machine level optimizations and machine code compilation. Why do we want to do all of it on every client, if not for every request, when we can do most of the work once and on the server-side?

And here comes the solution. WebAssembly addresses this exact problem. It’s a binary executable format for browsers. It promises to make everything better, faster, more robust, more reliable, more maintainable. And it also makes JavaScript irrelevant.

Now every language that can compile to WebAssembly, which is basically every compilable language ever, will work for front-end development. Java, Python, C#, C++, even Fortran if you’d like. So JavaScript would not be the only option soon enough. The very first time in its existence it would face competition in its own domain.

Not that the competition will necessarily kill JavaScript. Technically, Objective-C was not killed by Swift. It is still quite popular, just not like in the good old days. And Delphi wasn’t exactly slaughtered by C#. And Lisp machines didn’t entirely lose to Unix all at once. And, you might not believe me, but there is still rather active development going in Fortran as well.

You should only realize, that every career choice is a gamble. Nothing is for sure. And nothing is for life. Well, unless you’re going to live a really short one. JavaScript is on top of its game right now. It looks big and shiny and promising. But technologies change. Everything does.

WebAssembly will definitely be a huge game-changer for the JavaScript world. And I’m not even sure, there will be a JavaScript world some 10 or 20 years from now.

You don't have to learn assembly to read disassembly

Reading disassembly is more like reading tracks than reading a book. To read a book you have to know the language but reading tracks, although it gets better with skills and experience, mostly requires attentiveness and logical thinking.

Most of the time we read disassembly only to answer one simple question: *does the compiler do what we expect it to do*? In 3 simple exercises, I'll show you that often enough you too can answer this question even if you have no previous knowledge of assembly. I'll use C++ as a source language, but what I'm trying to show is more or less universal, so it doesn't matter if you write in C or Swift, C#, or Rust. If you compile to any kind of machine code—you can benefit from understanding your compiler.

1. Compile-time computation

Any decent compiler tries to make your binary code not only correct but fast. This means doing as little work in run time as possible. Sometimes it can even conduct the whole computation in compile-time, so your machine code will only contain the pre-computed answer.

This source code defines the number of bits in a byte and returns the size of int in bits.

```
static int BITS_IN_BYTE = 8;

int main() {
    return sizeof(int)*BITS_IN_BYTE;
}
```

The compiler knows the size of an int. Let's say, for the target platform it is 4 bytes. We also set the number of bits in a byte explicitly. Since all we want is a simple multiplication, and both numbers are known during the compilation, a compiler can simply compute the resulting number itself instead of generating the code that computes the same number each time it's being executed.

Although, this is not something guaranteed by the standard. A compiler may or may not provide this optimization.

Now look at two possible disassemblies for this source code and deduce what variant does compile-time computation and what doesn’t.

```
BITS_IN_BYTE:                Main:
  .long 8                      mov eax, 32
main:                          ret
  mov eax, DWORD PTR
    BITS_IN_BYTE[rip]
  cdqe
  sal eax, 2
  ret
```

Of course, the one on the right does.

On 32-bit platform int's size is 4 bytes, which is 32 bits, which is exactly the number in the code.

You might not know that an integer function conventionally returns output in **eax** which is a register. There are quite a few registers but most important for us are the general purpose registers, more specifically **eax**, **ebx**, **ecx**, and **edx**. Their names respectively are: **a**ccumulator, **b**ase, **c**ounter, and **d**ata.

They are not necessarily interchangeable. You can think of them as ultrafast predefined variables of known size. For instance, rax is 64 bits long. The lower 32 bits of it is accessible by the name eax. The lower 16 bit of it as ax, which in its own turn consists of two bytes ah and al. These are all the parts of the same register. Registers do not reside in the RAM, so you can't read and write any register by the address.

The square brackets usually indicate some address manipulation. E. g.

```
mov rax, dword ptr [BITS_IN_BYTE]
```

This reads as “`put whatever` **lives by the address of** BITS_IN_BYTE **in the rax register** **as a double word**”.

The code on the right already has an answer in it, so it doesn't even need address manipulations.

2. Function inlining

Calling a function implies some overhead by preparing input data in the particular order; then starting the execution from another piece of memory; then preparing output data; and then returning back.

Not that it is all too slow but if you only want to call a function once, you don’t have to *actually call* the function. It just makes sense to copy or “inline” the function's body to the place it is called from and skip all the formalities. Compilers can often do this for you so you don't even have to bother.

If the compiler makes such an optimization, this code:

```
inline int square(int x)  {
    return x * x;
}

int main(int argc, char** argv)  {
    return square(argc);
}
```

Virtually becomes this:

```
// not really a source code, just explaining the idea
int main(int argc, char** argv)  {
    return argc * argc;
}
```

But the standard does not promise that all the functions marked as inline shall get inlined. It's more of a suggestion than a directive.

Now look at these two disassembly variants below and choose the one where the function gets inlined after all.

```
main:                        square(int):
  imul edi, edi                imul edi, edi
  mov eax, edi                 mov eax, edi
  ret                          ret
                             main:
                               sub rsp, 8
                               call square(int)
                               add rsp, 8
                               ret
```

Not really a mystery either. It’s the one on the left.

You might not even know, that the instruction to call a function is really called the **call**. It stores a special register that points to the next instruction after the call in the stack and then sets it to the function's address. A processor hence jumps to run the function. The function then uses **ret** to get a stored address from the stack back to the register, and make the processor jump back to from where it has been called.

Since the disassembly on the left doesn't even contain any recall of square, the function has to be inlined anyway.

3. Loop unrolling

Just like calling functions, going in loops implies some overhead. You have to increment the counter; then compare it against some number; then jump back to the loop's beginning.

Compilers know that in some context it is more effective to unroll the loop. It means that some piece of code will actually be repeated several times in a row instead of messing with the counter comparison and jumping here and there.

Let's say we have this piece of code:

```
int main(int argc, char**) {
    int result = 1;
    for(int i = 0; i < 3; ++i)
         result *= argc;
    return result;
}
```

The compiler has all the reasons to unroll such a simple loop, but it might as well choose not to.

Which disassembly has the unrolled loop?

```
main:
  mov eax, 1
  mov ecx, 3
.LBB0_1:
  imul eax, edi
  dec ecx
  jne .LBB0_1
  ret
```

```
main:
  mov eax, edi
  imul eax, edi
  imul eax, edi
  ret
```

It's the one on the right.

You don’t have to know that **j<*>** is the family of jump instructions. There is one unconditional jump **jmp**, and a bunch of conditional jumps like: **jz**—jump when zero; **jg**—jump when greater; or, like in our code, **jne**—jump when not equal. These react to the Boolean flags previously set by the processor.

The code on the right clearly has a repeating pattern, while the one on the left has a number three that is the loop's exit condition, and that could be enough to make a conclusion.

Conclusion

You can argue that these examples were deliberately simplified. That these are not some real-life examples. This is true to some degree. I refined them to be more demonstrative. But conceptually they are all taken from my own practice.

Using static dispatch instead of dynamic made my image processing pipeline up to 5 times faster. Repairing broken inlining helped to win back 50% of the performance for an edge-to-edge distance function. And changing counter type to enable loop unrolling won me about 10% performance gain on matrix transformations, which is not much, but since all it took to achieve was simply changing short int to size_t in one place, I think of is as a good return of investment.

Apparently, old versions of MSVC fail to unroll loops with counters of non-native type. Who would have thought? Well, even if you know this particular quirk, you can't possibly know every other quirk of every compiler out there, so looking at disassembly once in a while might be good for you.

And you don't even have to spend years learning every assembly dialect. Reading disassembly is often easier than it looks.

Fortran is still a thing

In 2017 NASA announced a code optimization competition only to cancel it shortly after. The rules were simple. There is a Navier-Stokes equations solver used to model aerodynamics, and basically, the one who makes it run the fastest on the Pleiades supercomputer wins the first prize.

There were a few caveats though. The applicant had to be a US citizen at least 18 years of age, and the code to optimize had to be in Fortran.

Fortran is not the most popular language in the programmer’s community. Many believe that it’s complex and outdated. Programming in Fortran is perceived as riding a horse-driven carriage to work. Fortran? Isn’t it something that Amish do?

Many thought that the competition will never start due to the lack of applicants. In fact, it was canceled for the exact opposite reason.

Quoting NASA's Press Release:

> *«The extremely high number of applicants, more than 1,800, coupled with the difficulty in satisfying the extensive vetting requirements to control the public distribution of the software made it unlikely we would achieve the challenge’s original objectives in a timely manner.»*

Marco Librero, NASA Ames Research Center [Public domain],
via Wikimedia Commons

Fortran is not, of course, outdated, and it’s not at all complex. In fact, it has grown into these myths exactly because it is that good at what it does. It was designed to make number-crunching easy and efficient. Its users are scientists and engineers; not computer scientists and software engineers, but the real ones. And when engineers have a tool for the problem, they solve the problem with the tool. The problem comes first, not the code.

That’s how the complexity myth has started. Most of the code you see in Fortran is indeed complex. But this is because:

a) the problem behind it is complex;

b) the code was written by domain specialists and not programmers.

Fortran is simple enough for domain specialists to write bad code and achieve good results with it.

And as for the outdatedness, the current Fortran standard is Fortran 2018. Fortran is constantly evolving one little step after another. The biggest holding factor here is its mission. It must remain simple for real engineers.

However, the modern Fortran already has modules, objects, generics, and built-in support for parallel computing. The foot shooting area grows steadily to meet the fiats of the modern world.

It still excels in the good old structured programming. It has features that mainstream C-like languages lack. For instance, it can exit or continue a loop from a nested loop.

```
rows: do i = 1, 10
  columns: do j = 1, 10
    if (a(i, j) == 0.) exit rows
    ...
  enddo columns
enddo rows
```

If has *case* statement with ranges.

```
integer :: temp_c

! Temperature in Celsius!
select case (temp_c)
case (:-1)
  write (*,*) 'Below freezing'
case (0)
  write (*,*) 'Freezing point'
case (1:20)
  write (*,*) 'It is cool'
case (21:33)
  write (*,*) 'It is warm'
case (34:)
 write (*,*) 'This is Texas!'
end select
```

And it can use an array of indexes to access another array.

```
real, dimension(5) :: a = [ 2, 4, 6, 8, 10 ]
integer, dimension(2) :: i = [ 2, 4 ]

print *, a(i)   ! prints 4. 8.
```

There is a brief introduction into modern Fortran by Lars Koesterke. If you think Fortran hasn't changed since the 60s, please look through it. You might get surprised.

So Fortran is not complex and it's not outdated. Now how come it went underground in the modern world?

The answer here is actually the same as for "where do old programmers go?" They mostly stay in business, but according to the Martin's lawn, as the number of programmers doubles every five years, they get vastly outnumbered by the younger folks. They just dissolve in a mass. But this is endemic for the software industry only. Fortran users are mostly the real engineers and their numbers don't grow exponentially. There is more or less the same amount of Fortran users, old and young, as it was some 30 or 40 years ago when Fortran was at its peak.

In a way, it has been at its peak all that time. It's the world around that has gone crazy. Languages come and go, paradigms ripe and rot, the whole software business is a part of the fashion industry now. But when you want your plane to fly, you still need to do the maths with your trusty Fortran.

That's the beauty of the things that work; they don't have to change much.

Learn you a Lisp in 0 minutes

But why?

Learning a language, you are not going to write in professionally, is like visiting a country you are not going to move in to. It may be tiring, but it’s fun, educational and it makes you appreciate other cultures. And Lisp is particularly fascinating to learn because of its influence on modern programming. You might see traces of Lisp in the most unexpected technologies like WebAssembly or GCC internal representation.

The only reason not to learn Lisp, or any other language, is the amount of effort it usually takes. But! If you only want to know the very basics of Lisp, you wouldn’t have to spend any effort at all.

But how?

Since you are reading this article, you probably know English. And this also means that you know a bit of French as well. Words like “concept”, “culture”, “action”, “instinct”, “machine”, “science”, and many more are actually shared between the two languages. Words like these are called “cognates”, and the word “cognate” is almost a cognate itself.

This happens because of long-lasting French influence on the English language. And the same goes for Lisp too. Its ideas and concepts are so widespread among modern languages that if you have any substantial experience in programming at all—you automatically know some Lisp.

So?

I've prepared several exercises so you could prove to yourself that you do know a little Lisp. As many languages and dialects belong to the Lisp family, I should specify that this quiz is based on WeScheme.

The exercises start with very simple things and gradually progress into obscurity. It’s ok not to get all the answers right, some of them would only work for programmers with a functional programming background.

Amuse-toi!

1 Which number would this evaluate to?

```
(+ 2 2)
```

2 Does this return true or false?

```
(= (+ 2 2) (* 2 2))
```

3 Is it a, b or c?

```
(first (list 'a 'b 'c))
```

4 Would it print “Apples” or “Oranges”?

```
(define apples 5)
(define oranges 6)
(if (< apples oranges)
    (printf "Apples")
    (printf "Oranges"))
```

5 What number would it be?

```
(define (dbl x)
  (* 2 x))

(dbl 2)
```

6 What number will this result to?

```
(define (fact x)
  (if (<= x 1)
      1
      (* x (fact (- x 1)))))

(fact 3)
```

7 What would this function do to the list?

```
(define (qs xs)
  (if (empty? xs)
      (list )
      (let (
          (middle (first xs))
          (others (rest xs)))
        (let (
            (left (filter (lambda (x) (<= x middle)) others))
            (right (filter (lambda (x) (> x middle)) others)))
          (append (qs left) (cons middle (qs right)))))))

(qs (list 4 5 1 2 3))
```

That’s it. The answers are under the baguettes.

By jules / stonesoup (bocadillos-3) [CC BY 2.0], via Wikimedia Commons.

Well, I needed something long to screen the answers, so why not baguettes?

1. That’s just prefix notation for `2+2`, so the **answer is 4**.
2. It is **true**. `2+2 = 2*2`.
3. First element of list `(a, b, c)` **is a**.
4. Apples is just a variable set to 5, and oranges set to 6. `5<6` so it should print **`Apples!`**.
5. Function `dbl` is defined to double the argument. Then it’s called with `2` as an argument, so the **answer is 4**.
6. Function `fact` returns 1 for anything less than 1 and an argument `x` multiplied to `fact(x-1)`. So, `fact(0) = 1`, `fact(1) = 1*1`, `fact(2) = 2*1*1` and `fact(3) = 3*2*1*1` = **6**. **It is a factorial of 3** also known as "3!".
7. This is a simple quicksort implementation. The list gets split into pieces: some middle element, the “left” list with all the other elements lower than or equal to middle, and the “right” list with all the other elements greater than middle. “Left” and “right” get quicksorted recursively until they’re done and the function then returns `sorted left + middle + sorted right`. So the list `(4, 5, 6, 1, 2, 3)` **becomes sorted**: `(1, 2, 3, 4, 5, 6)`.

If you have read this far, congratulations! You now know you know Lisp!

If you got five answers right, you would probably be able to write a simple Gimp plugin on configure Emacs.

If you got all the answers then you have a knack for functional programming. If you haven’t been working in a functional language before, perhaps you should consider trying one.

Now what?

Of course, this test covers the very basics of Lisp. It doesn’t even touch the meta-programming, which is arguably the mightiest Lisp feature. But learning a bit of Lisp is not the whole point. The test actually shows that you should not stop at this, but learn more are more languages that you are not going to use professionally because it's easier than it seems.

Consider this. If practicing with JavaScript, or Python, or C#, or whatever your primary language is made you unknowingly learn some Lisp, then shouldn’t it work the other way around as well?

Eric Raymond once wrote:

> *“Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”*

But I don’t think Lisp is in any way magical. I think that learning any language enriches your experience. The more it’s different from what you do every day, the better. The more you know—the more ideas you have when approaching any new task. That’s the whole point of linguistic tourism.

If learning new languages, new concepts, and ideas, makes you a better programmer, then it is not just a waste of time, but an investment in your professional career.

N’est-ce pas?

Blood, sweat, and C++

I have a theory. But first of all I want you too see this:

```
#include <iostream>
int main() {
    int n = 3;
    int i = 0;
    switch (n % 2) {
    case 0:
    do {
    ++i;
    case 1: ++i;
    } while (--n > 0);
    }
    std::cout << i;
}
```

I found it on http://cppquiz.org/, which is by itself a great fun and a challenge. The program is valid, it is legit, it compiles, and it runs. It is an excellent bit for a quiz and an absolute crap of a code.

C++ is full of quiz material. It has enough obscurities for the whole professional life. It is just too inconsistent and overcomplicated. It combines 45 years of C legacy with all the misleads of modern programming.

In my humble fourteen years with C++ I found answers to a lot of questions. I know what integral promotions are, I know how slicing works, I know how come we have such a ridiculous switch statement. But there is one question, I never could answer properly before now. And it is: why are we doing this to ourselves?

There are better languages. Some are more mature, some are more novel. Some are more expressive, and some are more versatile. Some are mainstream, and some are esoteric. There are alternatives. But somehow C++ manages to keep its popularity, if not regain some in recent years. Why though?

My theory is, C++ programming plays the same strings as the Shackleton's ad.

In 1914 in order to recruit men for his Endurance expedition, Ernest Shackleton printed an advertisement in a newspaper and it goes like this.

> *Men wanted for hazardous journey. Low wages, bitter cold, long hours of complete darkness. Safe return doubtful. Honour and recognition in event of success.*

Needless to say, he found an excessive number of willing men instantly, and the advertisement has been since regarded as an exemplary piece of copywriting. Apparently, it is also a myth.

But C++ isn't. C++ is very very real. It's not as challenging as the Endurance expedition but it neither guarantees safe returns.

Perhaps, this is what drags smart people in. The very badness of a language becomes merit when you want to keep your mind excited in the day to day work. Java isn't nearly that dangerous, and C is not that variegated. These are excellent languages to get the job done, but do they provide the satisfaction of a good fight?

It probably starts in childhood, when you see your parents going to work and it almost seems like they disappear every day for nine hours or so. You know, that they do something there, you're not stupid, but it's all seems so boring and monotonous, it's almost like they voluntarily go out to non-existence day after day after day. And then when you grow older, you discover your first fear of non-existence, but you can't yet comprehend it, you just make choices that float you away from it. You need some challenge in life if only to prove that you don't just disappear every day for nine hours or so.

Some men go professional boxing; other join the army. Some choose C++.

If I were to invent a programming language for the 21st century

Quite a few programming languages were already invented in the 21st century. Swift, Kotlin, and Go are probably among the most popular. However, the distinctive feature of the 21st-century language design is the absence of any distinctive features in the languages themselves. The best thing about any of these is that you can spend a weekend and claim to learn a shiny new thing without actually learning anything new. They don't have anything new in them at all; they are all made by the "something done right" formula, this something being Objective-C, Java, or C.

While "not being new" is indeed a valuable trait in its own right, the question arises. Are they really the languages for the 21st century, or are they merely the reflection on the 20th-century bad programming habits?

If I were to invent a language, I wouldn't try to fix the past. I would try to invent a thing that not only works well in the reality of the modern world but can also evolve properly and stand the test of time. If this requires radical design decisions, so be it.

1. Down with the syntax!

Modern languages' syntaxes reflect the freedom of chalk and blackboard put into the shackles of ASCII. While some elements of notation like arithmetical signs and brackets are more or less idiomatic, some are just made up for no reason at all apart from saving the effort of pressing teletype buttons.

Typing is not an issue anymore. We are not obliged to play guess with our syntax.

Things like this[4]:

```
(($:@(<#[), (=#[), $:@(>#[)) ({~ ?@#)) ^: (1<#)
```

...are indeed concise and expressive. And also fun to write. But they do not help readability and, what's even more critical, googleability and stackoverflowability.

The same goes for cryptic function names, return code conventions, and attributes with obscure meaning. They served us well in the past, saving our punch-card space, now they deserve retirement.

Ultimately this:

```
FILE * test_file = fopen("test.txt", "w+");
```

Should become something like this:

```
create file test.txt for input and output as test_file
```

We don't need all those brackets, quotes, asterisks, and semicolons (unless they really help us to express things better). Syntax highlighting should work instead of syntax notation just fine.

Things that are cheap in the 21st century: parsing time, computer memory, online search. Things that are not: development time, programmer's memory, effort spent online learning the language specifics. This type of writing should facilitate the usage of cheaper things over the more expensive ones.

2. Down with the native types!

You probably know this as one of the JavaScript wats.

```
> 10.8 / 100
0.10800000000000001
```

It isn't, of course, JavaScript specific. In fact, it is not a wat at all, it is a perfectly correct behavior backed by the well-respected IEEE 754 standard. It's just how floating-point numbers are implemented in almost any architecture.

And it’s actually not that bad considering we are trying to squeeze an infinite amount of real numbers into 32, 64 or even 256 bits.

What mathematicians consider impossible, engineers do by trading off sanity for possibility. IEEE floating-point numbers are not, in fact, numbers at all. Maths requires real numbers' addition to have associativity. Floats and doubles do not always hold this property. Maths requires real numbers to include all the integers. This is not true even for the same sized *float* and *uint32_t*. Maths requires real numbers to have a zero element. Well, technically, in this regard IEEE standard exceeds the expectations, as floating-point numbers have not only one but two zero elements.

And it’s not only about floating-point numbers. Native integers are not much better. Do you know what happens when you add up two 16-bit integers like that?

```
0xFFFF + 0x0001
```

Well, nobody knows actually. Intuition tells, that the overflowed number should be *0x0000*. But this is not specified by any worldwide standard; it’s just how it usually goes with C and with x86-family processors. It may also result in *0xFFFF*, or trigger an interrupt, or store some special bit in a special place signaling that the overflow happened.

It is not specified at all. It differs. While floating-point numbers are just standardly insane, these are entirely unpredictable.

What I would propose for numeric computations instead is fixed point arbitrary sized data types with standard defined behavior on underflow, overflow, and precision loss. Something like this:

```
1.000 / 3.000 = 0.333
0001 + 9999 = overflowed 9999
0.001 / 2 = underflowed 0
```

Of course, you don’t have to actually write all the trailing zeros, they should be implied by the data type definition. But you should be able to select your maximum and minimum bounds for the type yourself, not just rely on the current processor's architecture.

Wouldn’t it work much slower then? Yes, it would. But realistically, how often do you have to program high-performance computations? I suppose unless you work in a narrow field of research and engineering that requires exactly that, not very often. And if you do, you have to use specialized hardware and compilers anyway. I’ll just presume, a typical 21st-century programmer don't have to solve differential equations very often.

That being said, shouldn’t fast, complex, and unpredictable native types from the past be an option and not the default?

3. Down with the metalanguaging!

There are brilliant wonderful languages designed not to do the task, but to create languages that do the task. Racket, Rebol, and Forth to name a few. I love them all, they are a pure delight to play with. But as you might guess, being fun is not exactly what makes a language universally popular.

Language leverage, the ability to create new sub-languages for the task, is a great power, and it pays vastly to have it when you work somewhere on an isolated island all on your own. Unfortunately, if you write code for other people to understand, you have to teach them your language along with the code. And that’s when it gets ugly.

People are generally interested in getting things done, not learning the language they’d have to forget anyway after things are done. For other people, learning your language is just an effort that would hardly pay off. Learning something common and standardized, however, is an investment for life. Therefore, people will rather reinvent your language in the standard terms than learn it. And there you go: countless dialects for the single domain; people arguing about aesthetics, ideology, architecture and all the things that are irrelevant; million lines of code being written just to be forgotten in months.

Lisp guys went through all of that in the 80s. They figured out that the more of the practical part of a language is standardized—the better. And they came up with Common Lisp.

And it’s huge. The INCITS 226–1994 standard consists of 1153 pages. This was only beaten by C++ ISO/IEC 14882:2011 standard with 1338 pages some 17 years after.

A programming language should not be that huge. Not at all. It’s just that it should have a decent standard library filled with all the goodies so people wouldn't have to reinvent them.

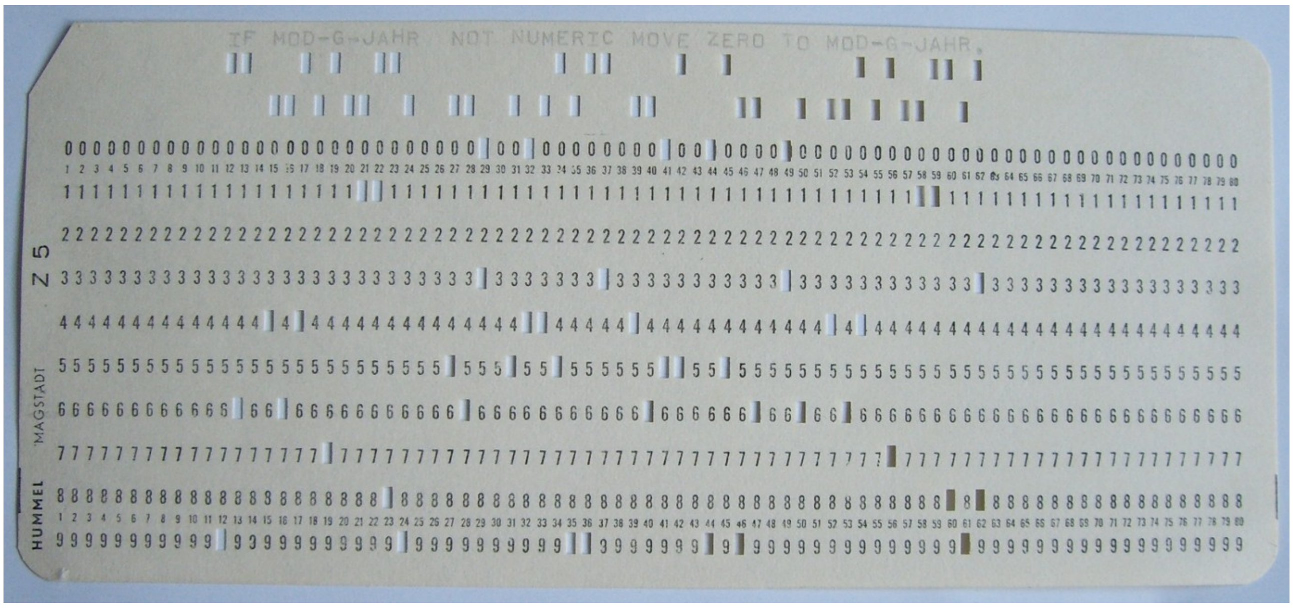
It is difficult to have both conciseness and applicability. With C++, we had to learn this the hard way. I think, to balance things properly, the language for the 21st century should be more domain-specific than not. Since business applications are currently the biggest mess, perhaps it should address that first and not some game development or web-design.

So…

The language for the 21st century should be business-oriented, English-like, and not dependent on any native types.

Wait a minute... Did I just reinvent COBOL?

Yes, I did.



```
IF MOD-G-JAHR  NOT NUMERIC MOVE ZERO TO MOD-G-JAHR.
```

By Rainer Gerhards (own work (own card, own photo))
[GFDL, CC-BY-SA-3.0 or CC BY-SA 2.5–2.0–1.0], via Wikimedia Commons

I have to confess, I deliberately described ancient COBOL's features as ultra-modern and super-promising to show you one thing. Language features don’t write the code. You do.

It’s naïve to think that the language is responsible for the quality of code and that by adding some bells and whistles (or removing some bells and whistles), we can automatically make everything better. We were not happy with Fortran and COBOL, so we invented C++ and Java only to be unhappy with them too in a few decades.

I feel like the core issue lies not in the field of programming. Are we really unhappy with the languages? Aren’t we unhappy with what we have done in general? How satisfied are you with the current state of software on a scale from 1 to -128?

Windows is vulnerable, Visual Studio is sluggish, and Vim is impossible to quit from. It takes 5 MB of traffic to download a page of text and 100 MB of RAM to display it. Desktop products are drowning with features nobody wants, gaming industry survives on patches, and don’t even get me started on banking software.

Somehow, with all the best practices, with all the methodologies, and with all the architectural breakthroughs we have acquired through the years, the software is only becoming worse. And that is truly disappointing, not the programming languages.

We have to blame something in our frustration. Being software engineers partially responsible for the world of crappy software, we wouldn’t blame ourselves, would we? So let’s blame the tools instead! Let’s reinvent COBOL again and again until one day we’ll invent the one true COBOL for everyone to be happy with and we will all become happy accordingly.

Probably not going to happen.

So if I were to reinvent a programming language for the 21st century, I would reinvent being responsible instead. I would reinvent learning your tools; I would reinvent being attentive to essential details and being merciless to accidental complexity. Unlike the languages that come and go with fashion, the things that matter do deserve constant reinvention.

Notes

1. As someone on Reddit noted, it's ironic that every answer for the C quiz is D. D or Dlang is the C++ done right. It has been designed by Walter Bright and Anrei Alexandrescu and it is just as brilliant as these two gentlemen are.

2. All the APL snippets in the book were run on the Dyalog APL available online at https://tryapl.org. There is also a free GNU APL at https://www.gnu.org/software/apl and there are open source implementations of derivative languages such as KONA: https://kona.github.io.

3. There is a bidirectional unambiguous correspondence between computer programs and mathematical proofs. It has a fancy name—Curry–Howard isomorphism. The invisible Prolog is, of course, just a metaphor.

4. It is a real piece of code written in a real language. It's a quicksort in J. J is another language designed by Kenneth E. Iverson. In a way, it's a re-invention of APL 25 years after. You might not believe me again, but it's even simpler than APL once you get to know it.

One final note.

This is a free e-book. It doesn't advertise anything. I get no profit from it.

I know that some people are uncomfortable with getting something without giving anything in return. If this is the case, please give me your feedback.

I'd appreciate any kind of feedback. Positive feedback powers me to keep writing and negative feedback pushes me into doing it better.

Write a tweet mentioning @wordsandbuttons; or a blog post with just a few lines and a link to https://wordsandbuttons.online/SYTYKC.pdf. That's how I'd know how to find it.

Or write to me directly: ok@wordsandbuttons.online.

Waiting to hear from you,

Oleksandr Kaleniuk