C++ Club UK Meeting 145

Gleb Dolgich

2022-03-17

Contents

War . . . 2
P2300 is headed to C++26 . . . 2
Modern C++ Course from Bonn U . . . 2
Mold 1.1.1 released . . . 2
String-like parameter cheatsheet . . . 3
Specializations of variable templates can have different type . . . 3
Secure coding practices . . . 5
Comparing Floating-Point Numbers Is Tricky . . . 6
    Herbie . . . 7
    Binary number representation cheatsheet . . . 8
Xmake package management . . . 8
A replacement for `std::vector<bool>` . . . 8
An interesting clang-tidy bug . . . 9
`nft_ptr` . . . 10
More Twitter, to keep our spirits up . . . 11
Quote . . . 14

War

Discussing C++ while a war in Europe is raging feels surreal. I'm with the people of Ukraine who are experiencing this great tragedy, I hope they prevail, and I hope the Russian war criminals will be brought to justice. My parents are in Lithuania, and I'm very worried about the future.

P2300 is headed to C++26

P2300 `std::execution` is headed to C++26. The latest poll with the question "Advance P2300R5 to electronic polling to send it to LWG for C++26" resulted in strong consensus with just a single neutral vote and no votes against. You probably remember that on its way to C++23 the paper was met with strong objections from quite a few people. Maybe the latest poll reflects the fact that C++26 is a loo-o-o-ng way away and the paper can be ready in time.

Modern C++ Course from Bonn U

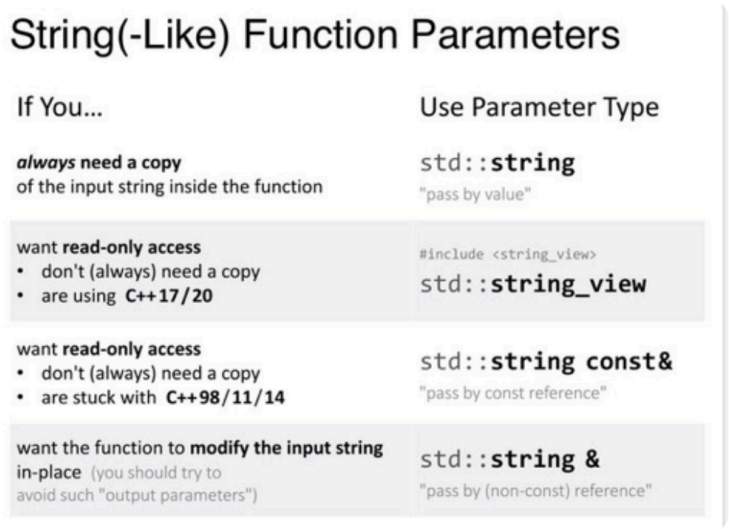
There is a decent Modern C++ course from Bonn University on YouTube, check it out.

Mold 1.1.1 released

Rui Ueyama released version 1.1.1 of his new fast linker mold. This version adds new LTO options for lld compatibility and reduces memory usage by ~6%, consuming less memory than GNU Gold or Clang lld.

String-like parameter cheatsheet

hacking C++ @hackingcpp
@fenbf
also for regular functions my rules are:



thread
1d • 12/12/2021 • 19:18

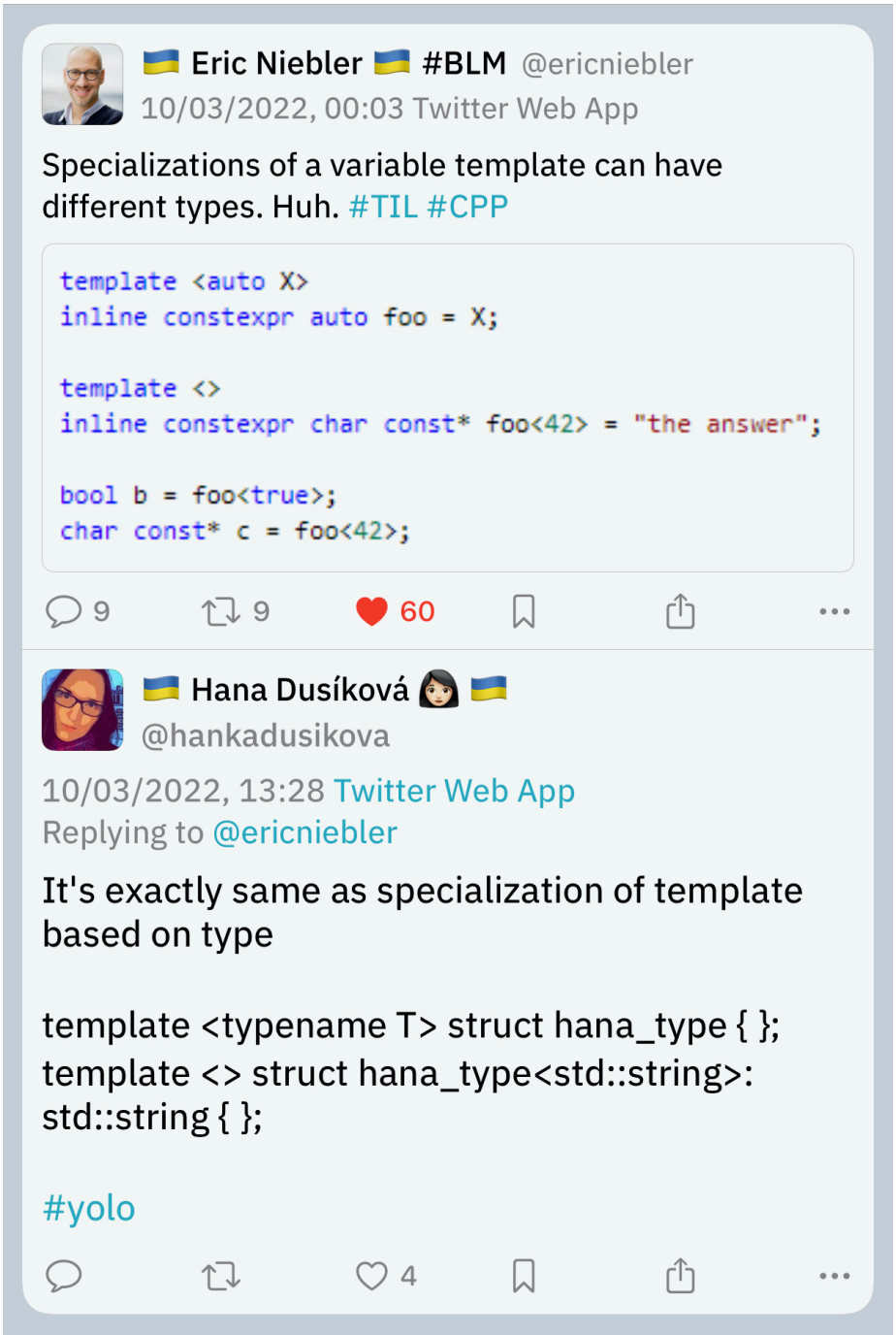
Specializations of variable templates can have different type

Eric Niebler tweets:

> Specializations of a variable template can have different types. Huh. #TIL #CPP

Hana Dusíková replies:

> It's exactly [the] same as specialization of [a] template based on type



Corentin Jabot follows up:

> One of the things that's currently bending my mind is that you can have a template variable which is a generic lambda

```
template <class>
auto x = []<class>{};
```

Secure coding practices

Amir Kirsh posted an article on the IncrediBuild blog called *Top 10 secure C++ coding practices*. In it he gives an overview of what security is and how a C++ programmer can make their code more robust to avoid vulnerabilities. He starts with the following:

> Understand that there are no safety nets provided by the compiler or runtime while coding in C++. C++ compiler generates the code the programmer asked it to generate, without adding any safety checks. While coding in C# or Java, for example, incorrect array access would lead to a runtime exception, whereas in C++ this leads to incorrect memory access or memory corruption in case of writing. Incorrect or sloppy coding can lead to overflows (stack, heap, and buffer overflows) which can easily be used for an attack.

Some of the advice from the author:

- Don't misuse APIs. Don't rely on undocumented behavior. Don't use APIs that are established to be vulnerable.
- Validate input.
- Take advantage of type safety. Don't intentionally bypass type checking.
- Be careful of arithmetic overflows and underflows. (*Ah yes, the infamous* `size_t`)
- Handle exceptions and errors carefully.
  - Don't leak sensitive information including error codes, stack traces, user IDs etc.
- Initialize variables.
- Security by obscurity is no security.
- Don't implement your own cryptography.
- Be careful with random numbers. Use the new C++11 random generators (*but not like that – see P0205*).
  - Don't use uninitialized variables as a random number generator (*What?*)
- Use C++ secure coding standard to complement your C++ coding standard, like SEI Cert C++.
- Use the right tools to detect security issues: static code analysers, sanitizers.

The related Reddit thread has an interesting discussion on using `at()` vs. `[]`. I didn't know that in some cases the compiler can optimize away bounds checks

in `at()`. Of course, a better solution is to use range-`for` loops or even better, ranges and algorithms.

Comparing Floating-Point Numbers Is Tricky

This is an old article from 2017 but it's still useful and provides a good illustration of the problems with machine representation of floating-point (FP) numbers.

Good things to remember:

- Floats cannot store arbitrary real numbers, or even arbitrary rational numbers.
- Since the equations are exponential, the distance on the number line between adjacent values increases (exponentially!) as you move away from zero.

Over the course of the article the author develops and improves a function to compare two FP numbers. He starts with this code which I've seen many times in our codebases, and explains why it's wrong:

```
1 bool almostEqual(float a, float b)
2 {
3     return fabs(a - b) <= FLT_EPSILON;
4 }
```

> We would hope that we're done here, but we would be wrong. A look at the language standards reveals that `FLT_EPSILON` is equal to the difference between 1.0 and the value that follows it. But as we noted before, float values aren't equidistant! For values smaller than 1, `FLT_EPSILON` quickly becomes too large to be useful. For values greater than 2, `FLT_EPSILON` is smaller than the distance between adjacent values, so `fabs(a - b)<= FLT_EPSILON` will always be `false`.

Boost has FP comparison API but the author explains how it is also not quite correct. He then arrives at ULPs:

> It would be nice to define comparisons in terms of something more concrete than arbitrary thresholds. Ideally, we would like to know the number of possible floating-point values—sometimes called *units of least precision*, or ULPs—between inputs. If I have some value `a`, and another value `b` is only two or three ULPs away, we can probably consider them equal, assuming some rounding error. Most importantly, this is true regardless of the distance between `a` and `b` on the number line.

The author emphasises the fact that ULPs don't work for comparing values close to zero, but this can be handled as a special case.

The main takeaways from the article are:

When comparing floating-point values, remember:

- `FLT_EPSILON...` isn't float epsilon, except in the ranges `[-2, -1]` and `[1, 2]`. The distance between adjacent values depends on the values in question.
- When comparing to some known value—especially zero or values near it—use a fixed epsilon value that makes sense for your calculations.
- When comparing non-zero values, some ULPs-based comparison is probably the best choice.
- When values could be anywhere on the number line, some hybrid of the two is needed. Choose epsilons carefully based on expected outputs.

This article was adapted from Bruce Davison's article *Comparing Floating Point Numbers, 2012 Edition*.

The GoogleTest macro `ASSERT_NEAR` uses a combination of ULPs- and epsilon-based comparisons and is the best way to compare FP values in tests against an epsilon difference.

David Goldberg's article *What Every Computer Scientist Should Know About Floating-Point Arithmetic* is a required reading for all programmers. A web-based version is here.

The corresponding Reddit thread is here.

A related article by John D. Cook, *Floating point numbers are a leaky abstraction*, points out a few cases when FP numbers don't behave as expected:

> Floating point numbers, the computer representations of real numbers, are leaky abstractions. They work remarkably well: you can usually pretend that a floating point type is a mathematical real number. But sometimes you can't. The abstraction leaks, though not very often.

Herbie

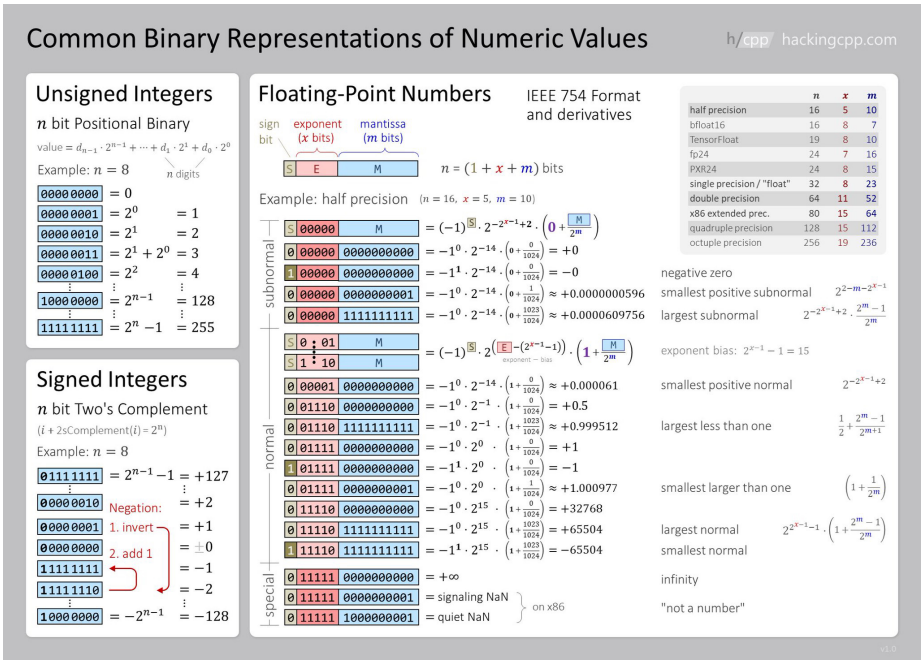
Herbie is a mind-blowing tool that simplifies arithmetic expressions to avoid FP issues.

> `sqrt(x+1)- sqrt(x)-> 1/(sqrt(x+1)+ sqrt(x))`
>
> Herbie detects inaccurate expressions and finds more accurate replacements. The [left] expression is inaccurate when `x > 1`; Herbie's replacement, [right], is accurate for all `x`.

Herbie can be installed locally or used from the web demo page. It is programmed in Racket which looks like a Lisp-like language.

Binary number representation cheatsheet

Source

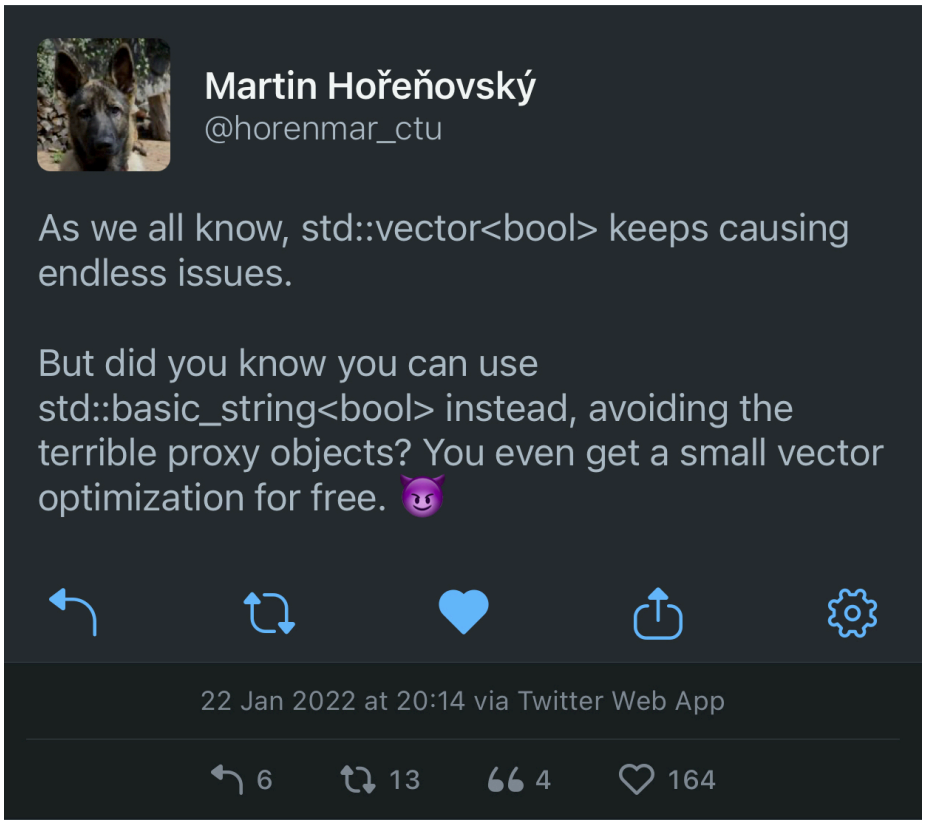


Xmake package management

This article describes package management in CMake using Vcpkg and Conan and compares it to what's available in Xmake. It also introduces Xmake's standalone package manager Xrepo. I'm still amazed at the quality and capabilities of Xmake. We lament about how difficult it is to bootstrap a C++ project, we have entire tools that create CMake project templates, but here it is, an easy to use and amazingly capable build system, and nobody seems to know about it. CMake is the standard, but teaching it to students is akin to starting C++ course by explaining pointers. Xmake could be an ideal student-friendly introduction to build systems at least for their toy projects, to avoid scaring them away before they even start learning C++.

A replacement for `std::vector<bool>`

Martin Hořeňovský tweeted:

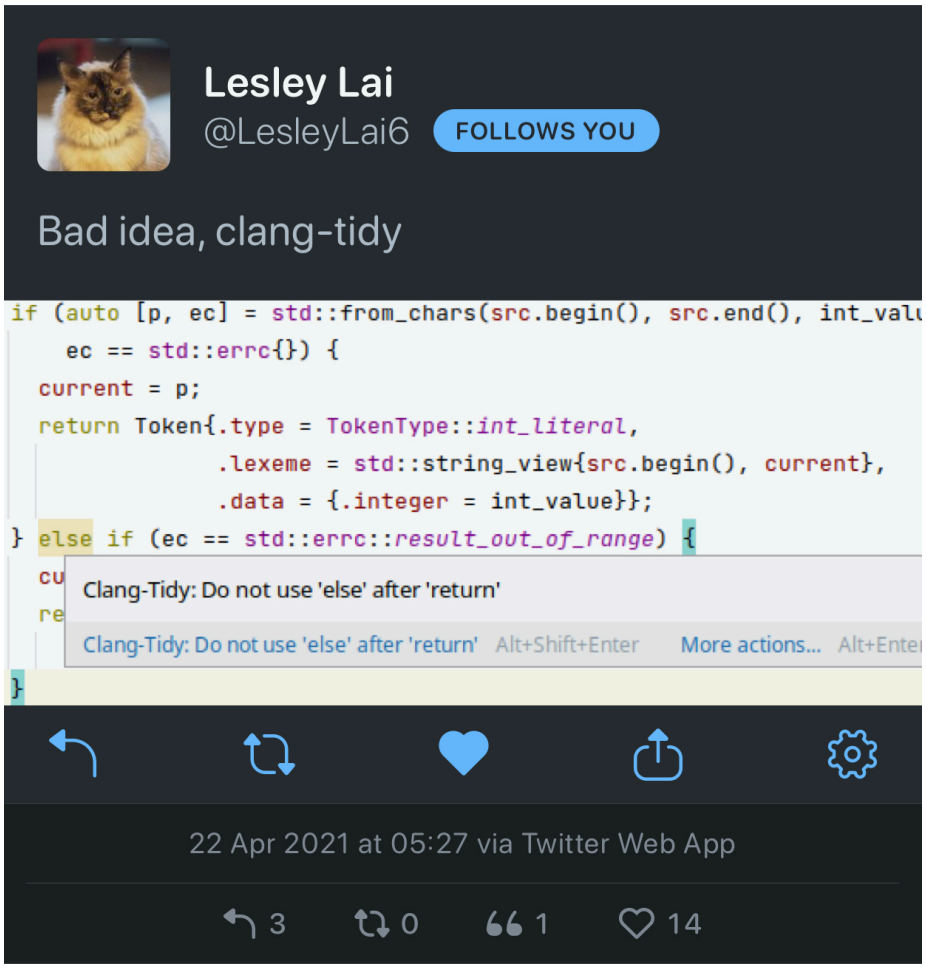


There is also a StackOverflow question about that. It seems like the idea might work but you shouldn't do it. Some replies:

- "Can somebody get people like this away from the keyboard, before they hurt themselves?" — Jan Wilmans
- "Somehow I equally love and hate this tweet." – Michail Caisse

An interesting clang-tidy bug

Lesley Lai tweeted:



If we applied the fix, the second `if` wouldn't compile because `ec` was declared in the first `if` init statement.

`nft_ptr`

Non-fungible tokens, or NFTs, are a scam built on the blockchain technology. There are many articles explaining this latest high-tech planet-destroying pyramid scheme, so I'm not going to do that here. Instead let me tell you about this excellent project that highlights the craziness from the C++ point of view. Behold `nft_ptr`: "C++ `std::unique_ptr` that represents each object as an NFT on the Ethereum blockchain."

```
1 auto ptr1 = make_nft<Cow>();
2 nft_ptr<Animal> ptr2;
3 ptr2 = std::move(ptr1);
```

> This transfers the Non-Fungible Token `0x7faa4bc09c90`, representing the Cow's memory address, from `ptr1` (OpenSea, Etherscan) to `ptr2` (OpenSea, Etherscan).

It works, and is completely bonkers. I especially like the **Why** section:

- C++ memory management is hard to understand, opaque, and not secure.
- As we all know, adding blockchain to a problem *automatically* makes it simple, transparent, and cryptographically secure.
- Thus, we extend `std::unique_ptr`, the most popular C++ smart pointer used for memory management, with blockchain support.
- Written in Rust for the hipster cred.
- Made with [love] by a Blockchain Expert who wrote like 100 lines of Solidity in 2017 (which didn't work).

The **Performance** section doesn't disappoint either:

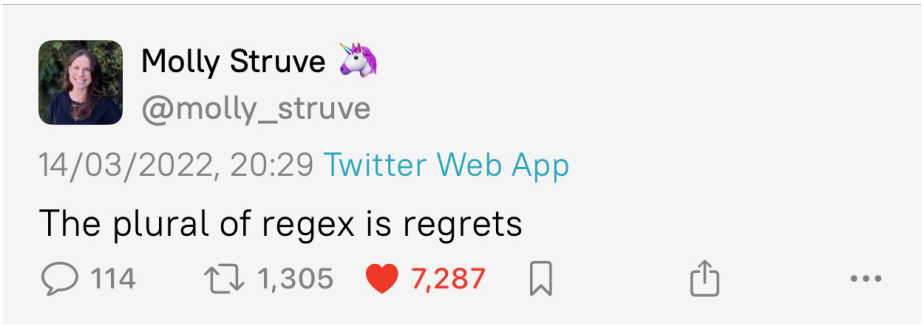
> `nft_ptr` has negligible performance overhead compared to `std::unique_ptr`, as shown by this benchmark on our example program:
>
> - `std::unique_ptr` - 0.005 seconds
> - `nft_ptr` - 3 minutes

The project is very thorough and even has a link to a whitepaper! It's indeed a white paper.

More Twitter, to keep our spirits up

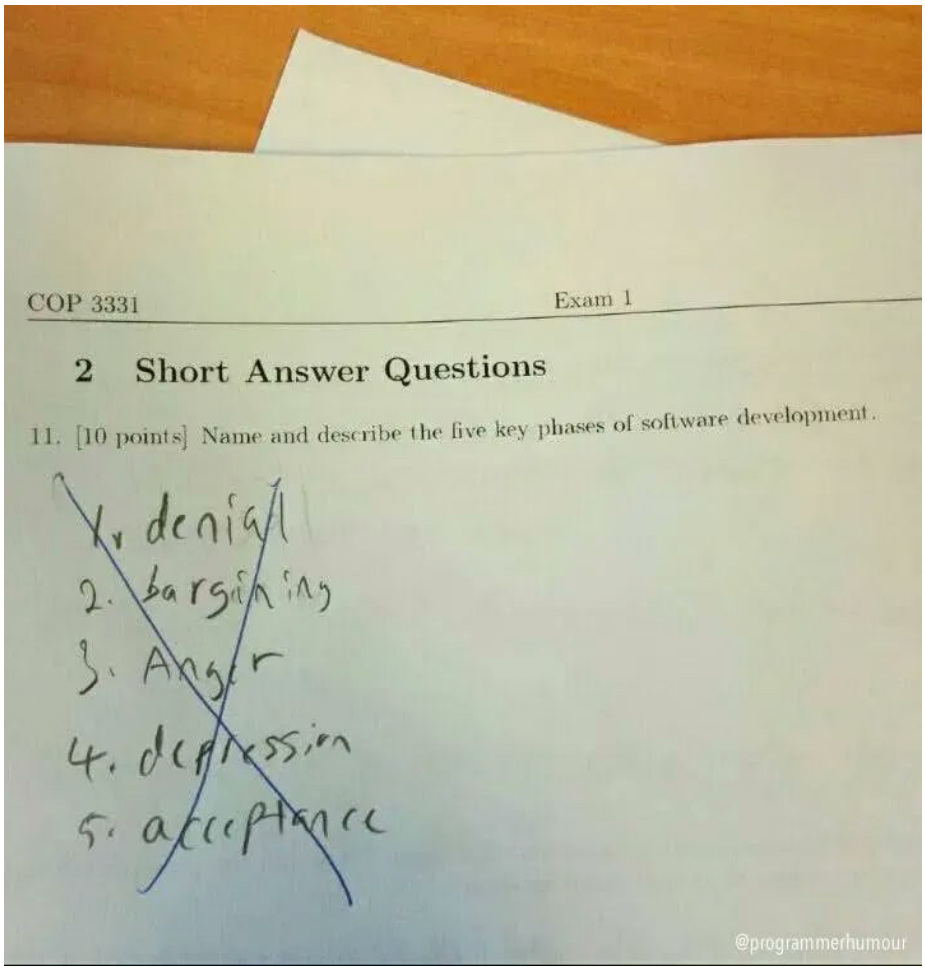
Vicki Boykis (@vboykis) tweets:

Molly Struve (@molly_struve):



TerraTech (@gaya_tech) on embedded programming:

An incorrect, apparently, exam answer to the question about phases of software development:



And finally, from Patricia Aas (@pati_gallardo):

Patricia Aas 🐢 @pati_gallardo

Computer Science is half-remembering something and googling the rest.

2y • 15/05/2019 • 18:01

Quote


Kyriakos Kapakoulak
September 1 ·
I learn from the mistakes of people who take my advice
· Hide Translation · Rate this translation