C++ Madness

Pitfalls, Inconsistencies and Arbitrary Nonsense

Dominik Charousset, February 2016

Why C++?

C++ combines the power of assembly languages

... with the readability of assembly languages

- Old saying

But it's still better than C?

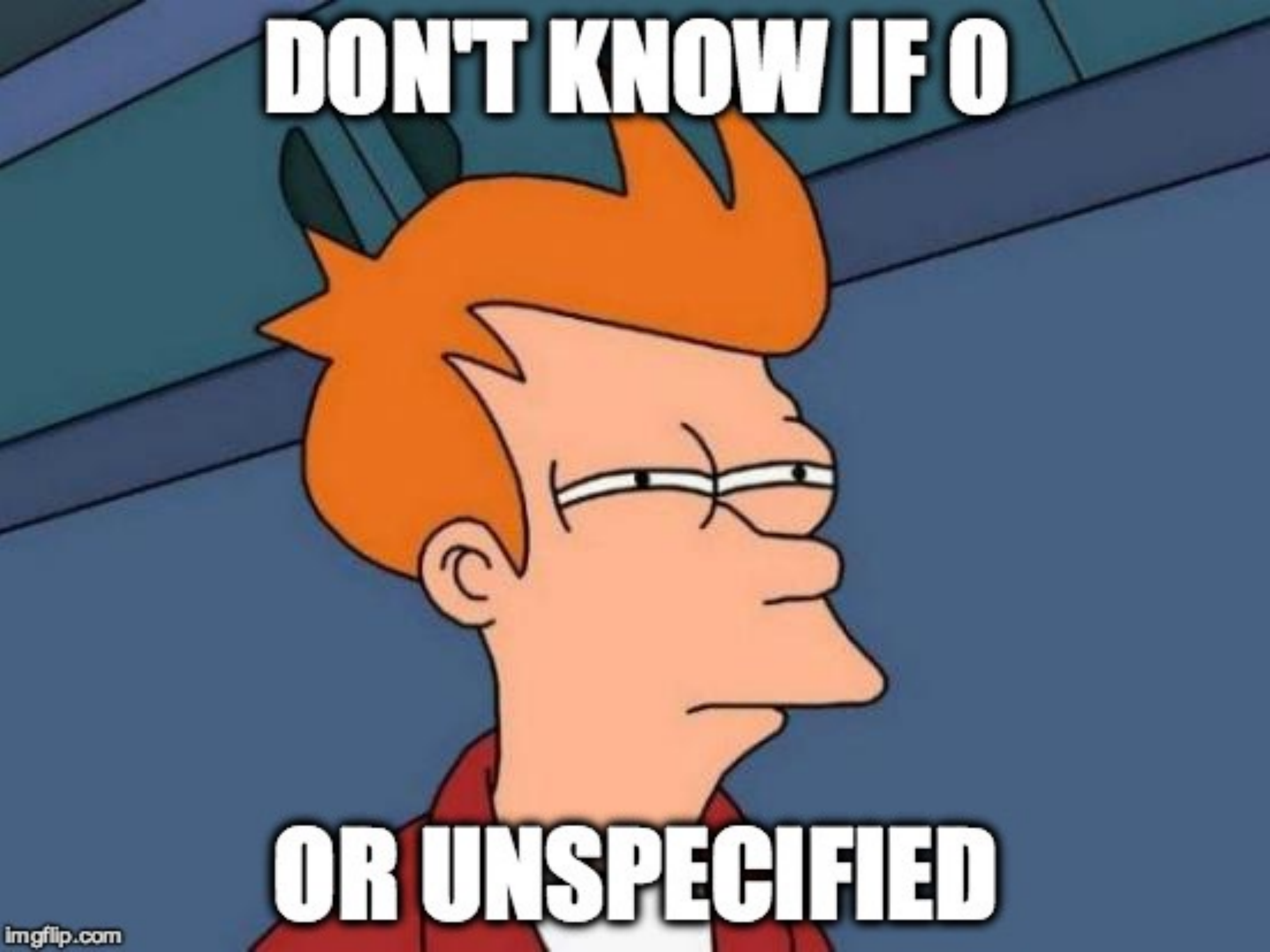
C makes it easy to shoot yourself in the foot;

C++ makes it harder, but when you do, it blows away your whole leg.

- Bjarne Stroustrup

Reasoning about C++?

- Consider: `int x;`
- What is the value of `x`?
  - `0`? `Unspecified`?
  - Obviously: *it depends!*


DON'T KNOW IF O
OR UNSPECIFIED
imgflip.com

Object Oriented?

- Plain old classes:

```
struct a {
  void foo();
};


struct b : a {
  void bar() {
    foo();
  }
};
```

- Does compile.

- Generalized classes:

```
template <class T>
struct a<T> {
  void foo();
};

template <class T>
struct b : a<T> {
  void bar() {
    foo();
  }
};
```

- Does not compile.

ONE DOES NOT SIMPLY

USE TEMPLATES IN C++

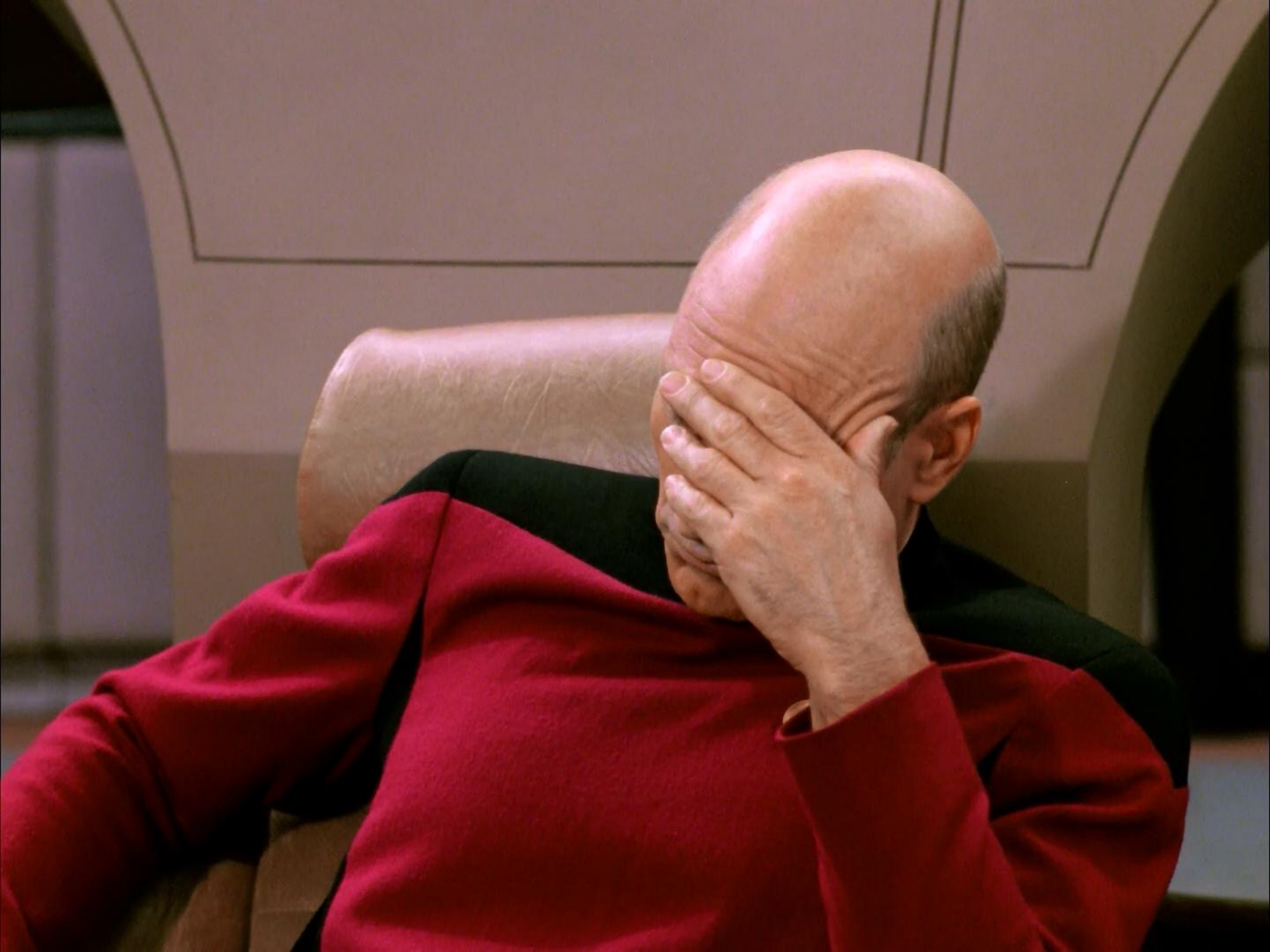
Case Study

- This code *runs fine* on Clang, *segfaults* on GCC

```
post_process_invoke_res(
    this, current_element_->mid.is_request(),
    fun(std::move(current_element_)));
```

- Why?

- C++ does not specify argument evaluation order!



Wait, I Lied!

- C++ *does* specify evaluation order.
- But only for initializer lists! Because... ***Reasons!***

Within the initializer-list of a braced-init-list, the initializer-clauses, including any that result from pack expansions (14.5.3), are **evaluated in the order in which they appear**. That is, every value computation and side effect associated with a given initializer-clause is sequenced before every value computation and side effect associated with any initializer-clause that follows it in the comma-separated list of the initializer-list. [Note: This evaluation ordering holds regardless of the semantics of the initialization; for example, it applies when the elements of the initializer-list are interpreted as arguments of a constructor call, **even though ordinarily there are no sequencing constraints on the arguments of a call**. —end note]

DOUBLE FACEPALM

When the Fail is so strong, one Facepalm is not enough.

Beyond Syntax

- Syntax is negligible, right?
- The standard library is where it's at!
  - High level building blocks for I/O.
  - Nice abstractions for concurrency & distribution.

High level I/O?

- C++ cannot serialize *its own objects*.

- No serialization API, no reflections, no nothing!

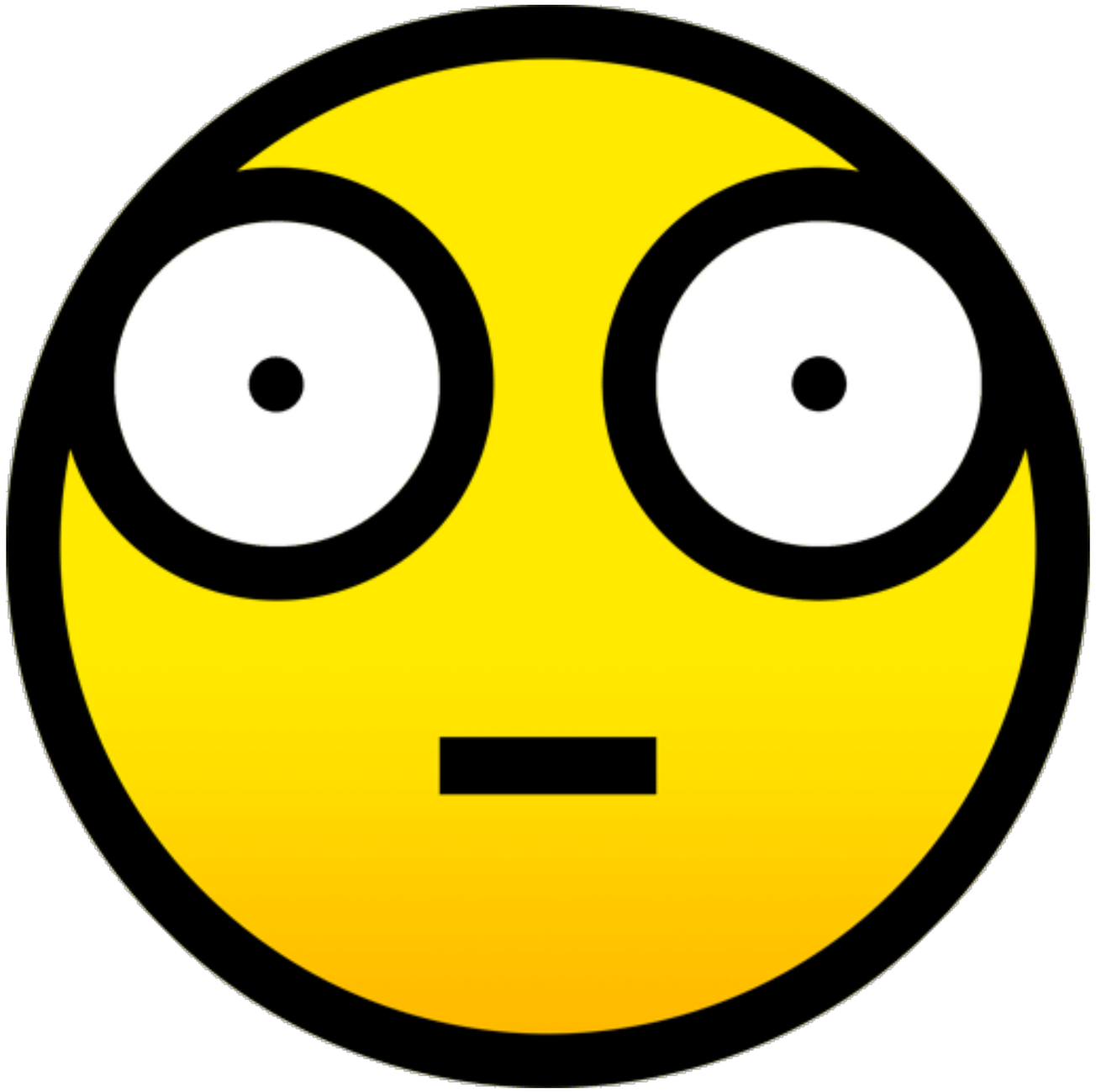
Concurrency is Easy!?

- No better task abstraction than `std::async`?
- Atomics: 6 different memory orders!
- Futures: not composable; not even `.then` exists!
- Mutexes: hello low-level, deadlock-ridden world...

Distribution Support?

- Nothing. Literally.

- Networks don't exist in C++ world (until C++17).



Why C++?

- It's full of arbitrary, often self-contradictory rules.
- Concurrency primitives are lackluster.
- Distribution is not even supported.

No Distribution, No Cry

- Who needs cloud computing, anyway?
- TCP/IP and HTTP are just another hype. Right?


I CAN HAS
NETWORK?
imgflip.com

Back to Sanity

- I will now stop bashing C++.
  - Although it really, really, *really* deserves it.
  - Scott Meyers is better at it anyways: https://www.youtube.com/watch?v=48kP_Ssg2eY
- Let us talk about something fun (and sane): Erlang

A Small Introduction to Erlang for C++ Devs

Dominik Charousset, February 2016

What has Erlang to Offer?

- Functional programming with dynamic typing.
- Message-oriented work flows.
- Scalable, multi-core friendly runtime (VM).
- Distribution support *built-in*!
- Pattern matching!


PATTERNS
PATTERNS EVERYHWERE
memegenerator.net

Yes, Patterns Everywhere!

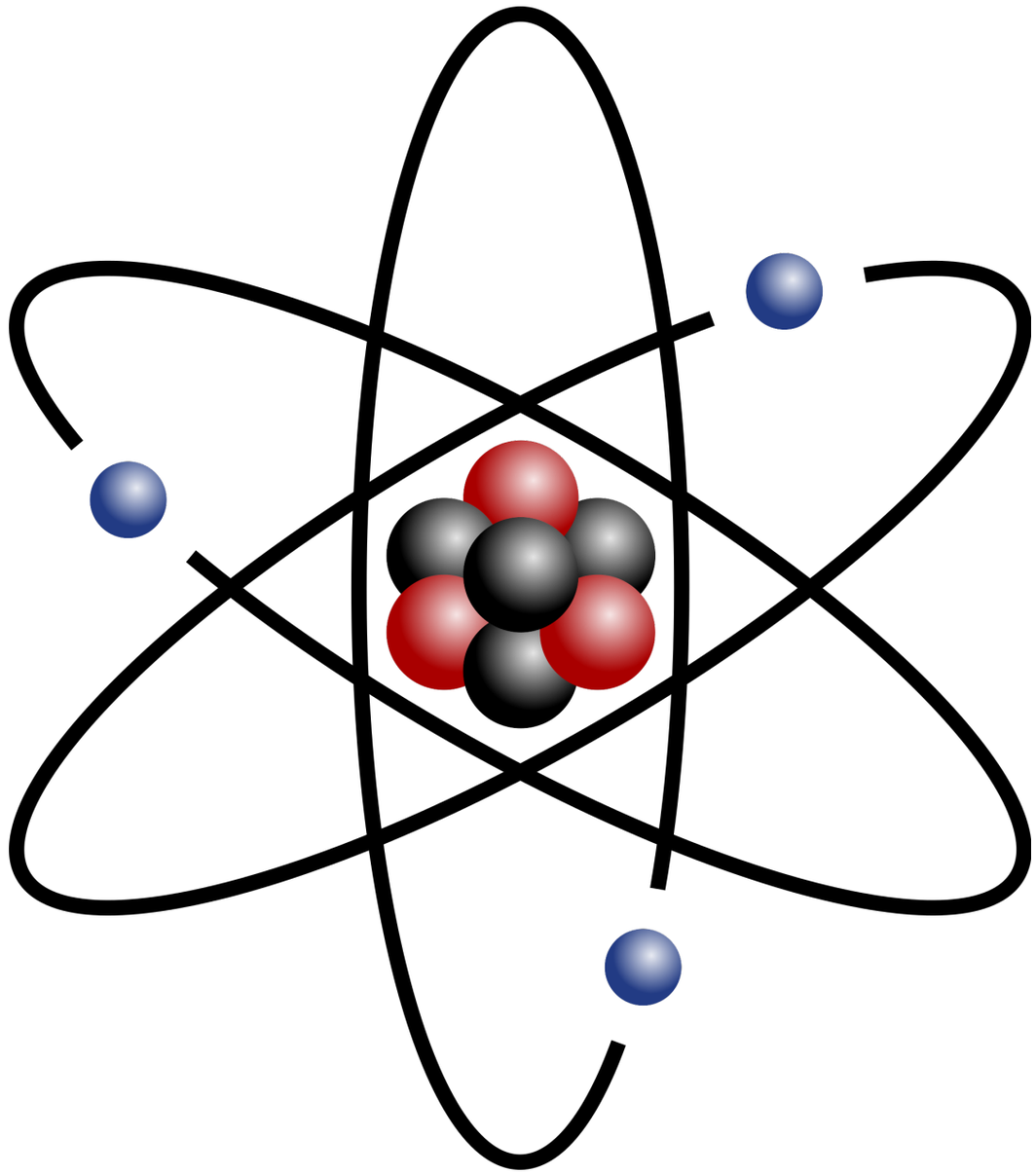
- Function dispatching? Patterns!
- Receive messages? Patterns!
- If-then-else? Patterns!
- ... you get the idea.

Erlang Syntax in a Nutshell

- Separate statements with ","
- Separate patterns with ";"
- End expressions with "."
- Start function names and atoms with lowercase.
- Start variable names with uppercase.

Atoms

- Constants.
- Start with lowercase or `'`
  - `my_atom`
  - `'My Atom'`
- Usually annotate data.

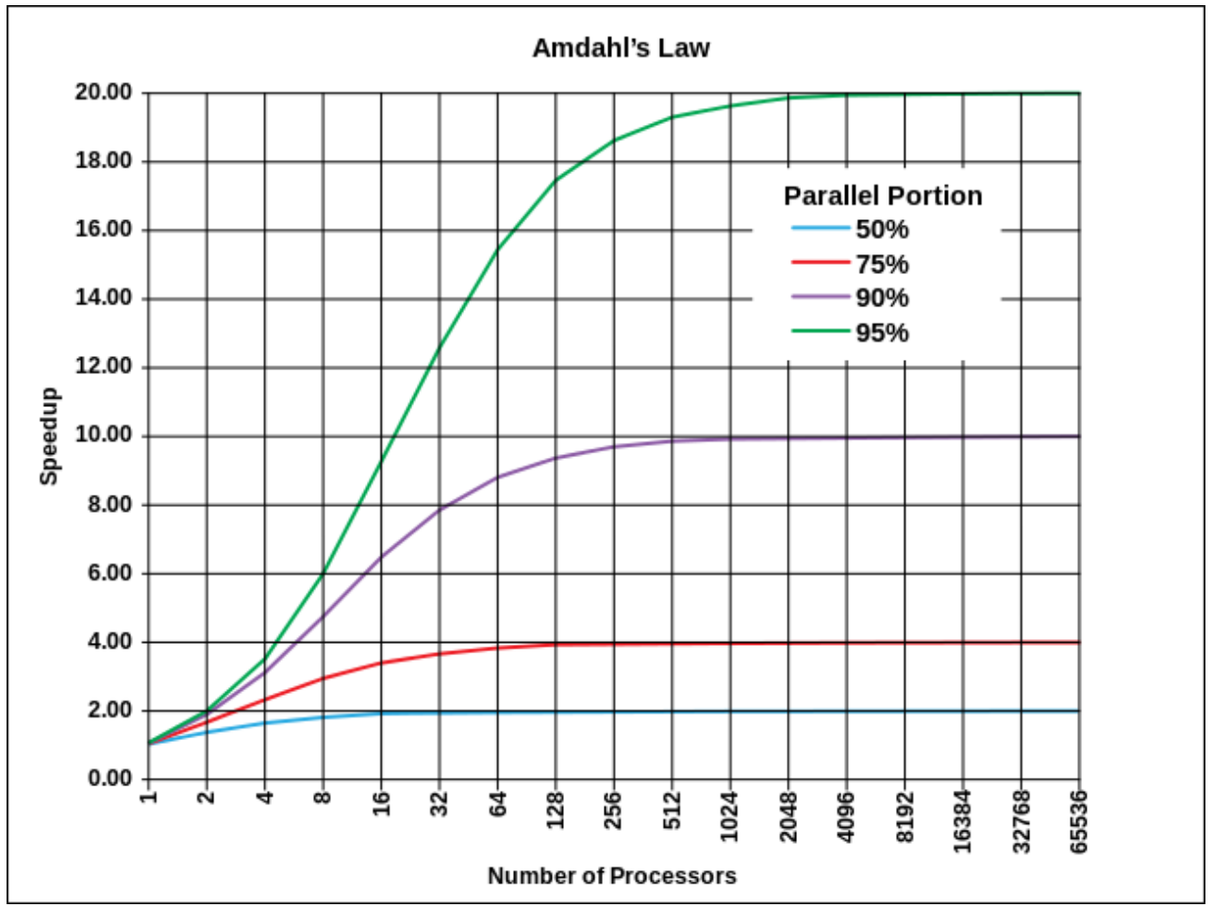


Simple Function

```
-module(fib).

-export([fib/1]).

fib(0) -> 0;

fib(1) -> 1;

fib(N) -> fib(N-1) + fib(N-2).
```

Demo Time

Amdahl's Law



Source: Wikipedia
https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg

- Speedup is limited by longest sequential part.
- Applications need to be split in many independent tasks.

Side Note: Actors

- Actors are computational entities that:

    1. Do not share state.
    2. Communicate via asynchronous messages.
    3. Can spawn (create) more actors.

- A process in Erlang essentially is an actor.

SPAWN

wn.com

Spawn (in Erlang)

- Starts new processes (aka actors).
  - Local: `spawn(module, fun, [args])`
  - Remote: `spawn(node, module, fun, [args])`

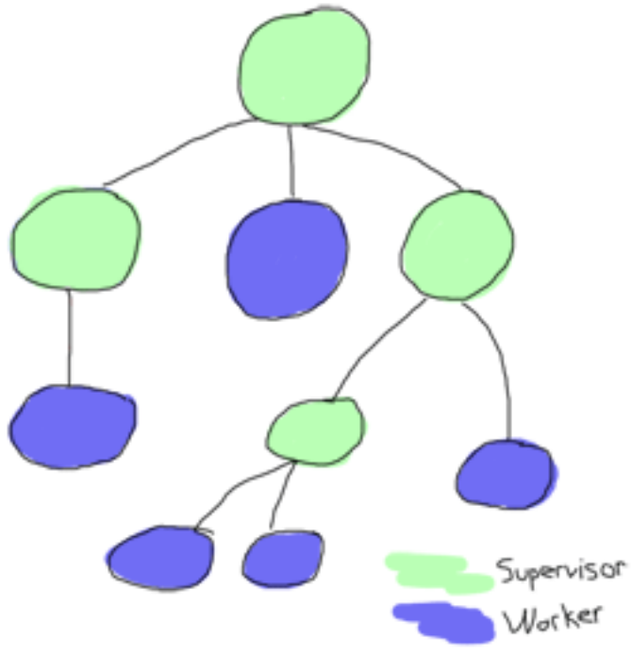
Message Passing

- Send (asynchronous) messages with "`! {args}`"
- Receive messages with "`receive ... end`"
- Transparently works locally and remotely!

Error Handling

- Actors can monitor other actors.
  - Receive a DOWN message if target fails.
    - Exit reason indicates source of error.
- Links are bidirectional monitors.
  - Fail for the same reason as link (non-normal exit).
  - Trap exits to override default behavior.

Supervision Tree



From: http://learnyousomeerlang.com/supervisors

- Hierarchic "failure tree".
- Each supervisor monitors its children.
- Restart policies for automatic re-deployment.

Ping Pong

```
ping(Pong) ->
 Pong ! {ping, self()},
 receive
  pong ->
   io:format("Got pong!~n")
 end.
```

```
pong() ->
 receive
  {ping, Ping} ->
   Ping ! pong,
   pong()
 end.
```

```
Pong = spawn(pingpong,
             pong, []),
Ping = spawn(pingpong,
             ping, [Pong]).
```

Demo Time


OTP
imgflip.com

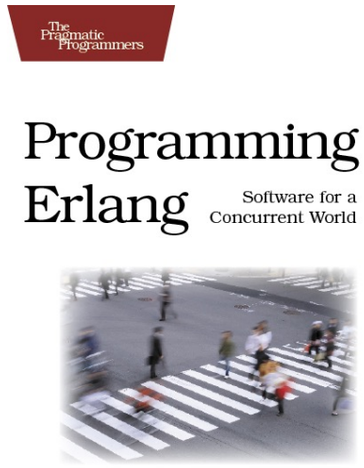
OTP (Open Telecom Platform)

- Building blocks for concurrent & distributed apps.
    - Abstraction for common behaviors.
        - Includes basic server/client templates, etc.
- The true reason to use Erlang.
- Would require its own talk to cover in depth.

Where to Start?

- http://learnyousomeerlang.com/
  - Amazing, free online guide.

- `erl`
  - Erlang's shell. Perfect for learn-by-doing!
- "Programming Erlang"
  - From the main designer of Erlang.



… Back to C++ Again?

- If you're a masochist like me, you'll go back to C++
- If you do, do yourself a favor*:
- Use the C++ Actor Framework (CAF): http://actor-framework.org
- Message-oriented programming makes concurrency and distribution so much easier.

* *disclaimer:* My opinion is highly biased, since I am a maintainer of CAF

Thanks for listening!

Any questions?