Instruction Reordering Everywhere:

The C++ **"As-If" Rule** *and the* **Role of Sequence**

charley bay
c.bay@f5.com
charleyb123@gmail.com

F5 Networks, Inc.

Two Talks

Today!

1 **Physical vs. Logical Sequence**

(undefined vs. defined behavior)

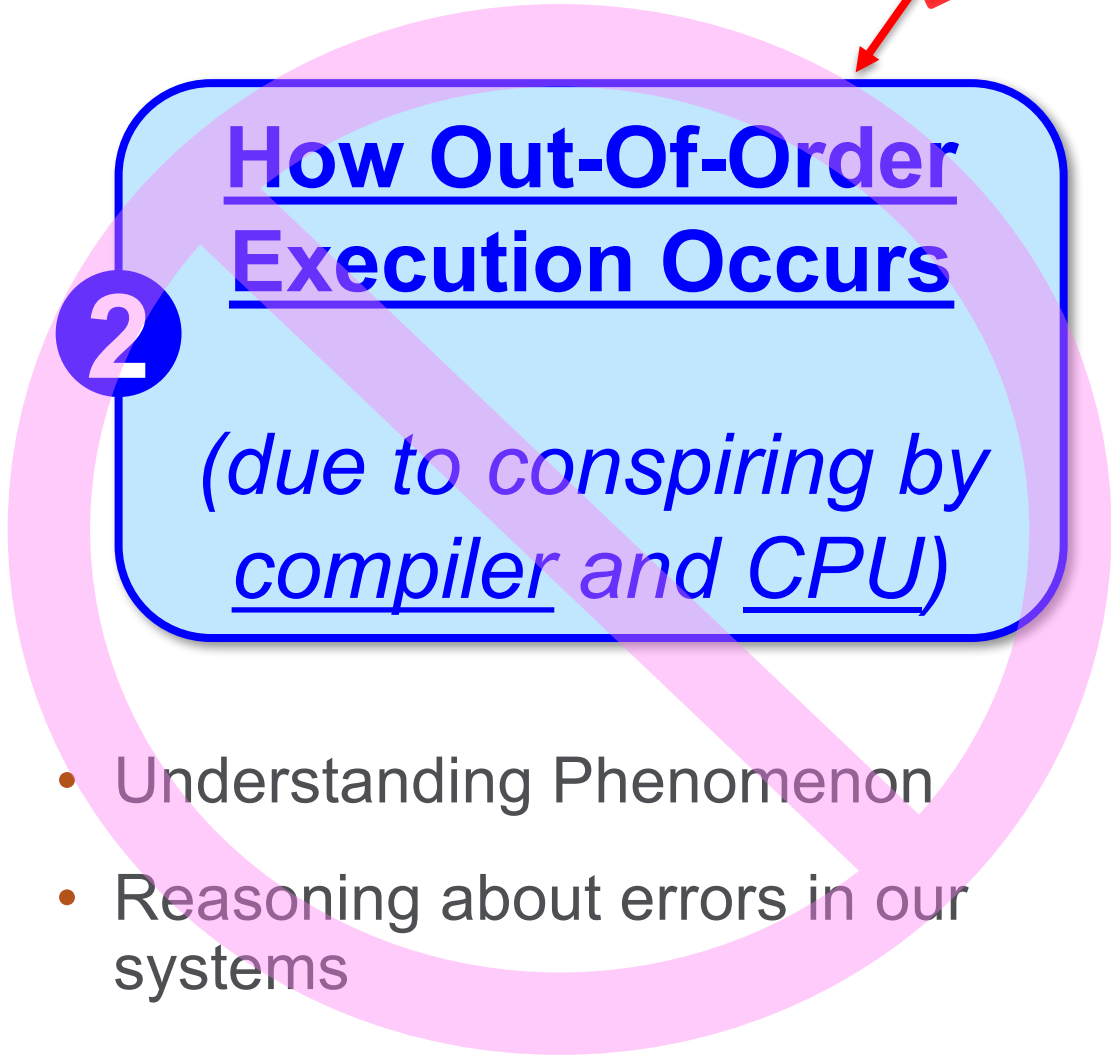
- How to think about dependencies
- How to design algorithms

NOT Today!

2 **How Out-Of-Order Execution Occurs**

(due to conspiring by compiler and CPU)

- Understanding Phenomenon
- Reasoning about errors in our systems



Not Today

An introductory (i.e., “**First Principles**”) dive into **instruction re-ordering**

(at **compile-time**, and at **run-time**)

due to conspiring by the **compiler and CPU** to make most efficient use of execution units and resources within the CPU processor core.

Focus on:

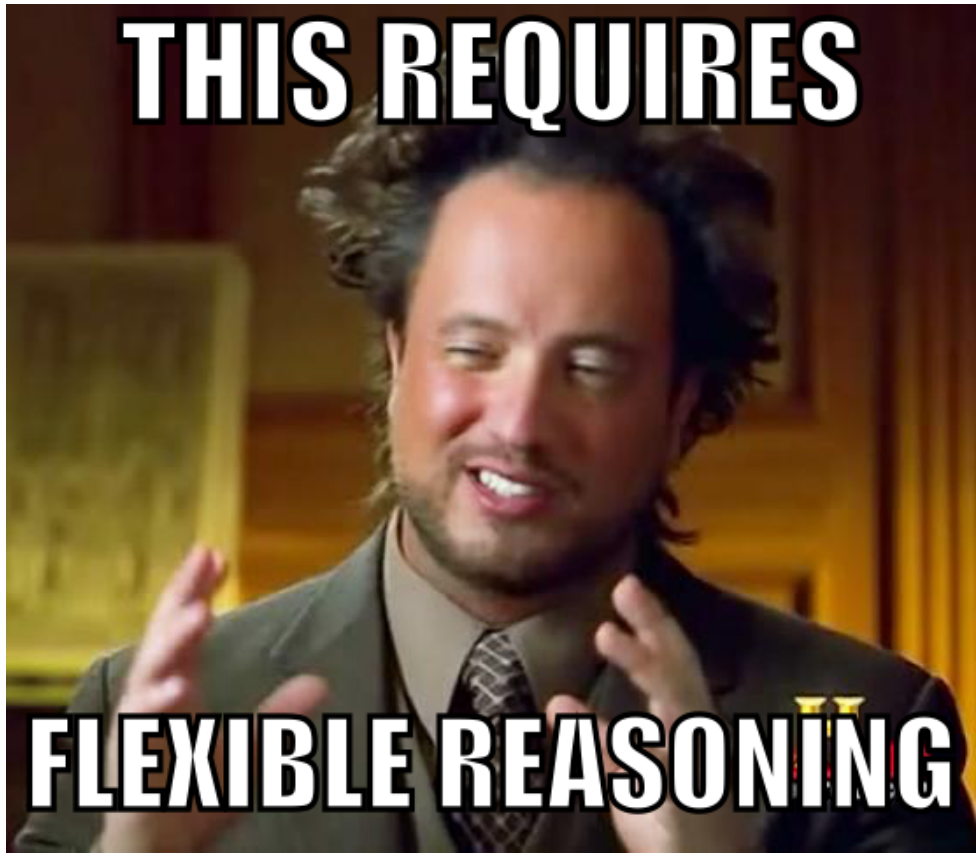
- Compiler Internals
- CPU Internals
- Runtime Execution

The CPU Cache: Instruction Re-Ordering Made Obvious

<https://www.youtube.com/watch?v=tNkVUIv2gEE>

Goals

1. Explain the importance of the **C++ "As-If" rule**
2. Trace **data-flows** and **control-flows** in algorithms
3. Discuss **Imperative vs. Sequential Devices**
4. Why Instruction Re-Ordering is **A "Good Thing"™**



Discussion:

Software Engineers can **be more "flexible"** in reliance on **imperative versus sequential** execution, and **physical versus logical sequences**.

The Question:

What is "**Sequence**", and *what is its* ***role***?

Today's Agenda

1. **Why Care** About **Sequence**? (Why *not*?)
2. (Order) **Assumptions In Design**
3. **Imperative vs. Sequential Devices**
4. **Physical vs. Logical Sequences**
5. The **C++ "As If" Rule**
6. **Summary**

Not time for today:

- **How Out-of-order execution occurs** *(answered through discussion of compiler and CPU internals)*

Take-Aways For Today

- We **cannot** control **instruction order**
  - *(is **not** well-defined!)*
- We **can** control **dependencies**
  - *(**are** well-defined!)*

With unbelievably high penalties, we can control some things; *but*

- we should not control order unless absolutely necessary; *and*
- we should prefer designs (algorithms) where we don't control order; *and*
- it's better in every way when we don't control order.

Focus On What's Important: **Dependencies**!

Take-Away:

- **Dependencies** … Worry!
- **Instruction Order** ... Don't Worry! *(Be Happy!)*

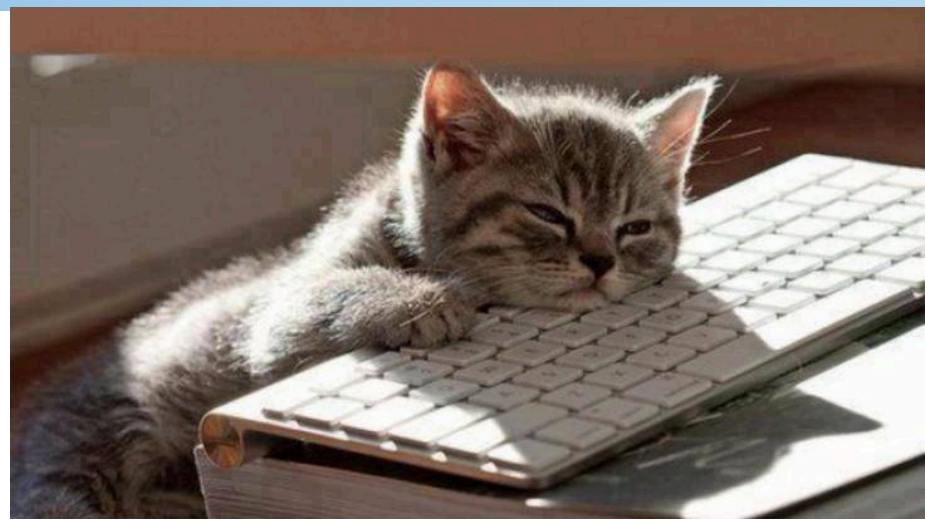
- We don't care about instruction order.
- We might think we do, but we don't, and shouldn't.

After Today...

- After Today, **You Will Know**:
  - My statements **executed in the CORRECT order**
  - My statements **executed in the EFFICIENT order**
  - My statements probably did **NOT** execute **in the order I specified**
- After Today, **You Will Be**:
  - **CONFIDENT** in your compiler
  - **TRUSTING** of your CPU
  - **GRATEFUL** you don't need to worry about instruction order
- **Moral Of The Story**:
  - It is **YOUR JOB to Design and Implement** with concern **for dependencies**
    - *We are Engineers! This is what we do! Single-threaded or Multi-threaded!*
  - It is a **Fool's Errand** to be concerned with **Instruction Order**
    - *Pointless! Distraction! Red-Herring!*

The Good News Take-Away:

"Be More Lazy And Ignorant!"



- You **should NOT need to know** any of this!
- You should **WORK** to **be MORE LAZY** regarding what you care about!
- You can **TRUST** the **CPU is always correct**!
  - And usually the **Compiler too**!

The More You "Know", the more likely you are hurting yourself and your algorithm

- …through **misplaced and incorrect assumptions**

You CANNOT possibly know the architectural and runtime-specific details! (THEY CHANGE!)

ANY assumption is EVENTUALLY violated!

(definition) **Assumption**: Any perceived truth that is uncontrolled in the design or implementation

There is no guarantee that statements will execute in the order that the C++ programmer specified.

- This is called, "**Instruction Re-ordering**"
- *Why?*
  - Compiler Optimizations
  - CPU pipelining

Toy Story (1995)

- All is *not* just Chaos!

The Big Lebowski (1998)

There ARE Rules!

(What Are The Rules?)

Rules: That Upon Which We Can Rely

- We **Trust To Be True**, *Or All Is Lost:*

1. C++ "Well Defined" Behavior
2. The Compiler Works
3. The CPU functioned properly
4. The Laws Of Physics

C++ "Well-Defined" Behavior is very clear, and "Real-World" Practical!

It Does! (*Usually!*)

It Does! (*Always!*)

Latencies exist! (*Always!*)

System Failures:
The Software Engineer **ASSUMED** something other than (1), (2), (3), (4)

The Office (2005-2013)

- **No** Other Rules Exist
  - If designing "ecosystems", final rule is "*stuff breaks*"

Why Care About Sequence?

Because we reason about procedural algorithms

We DO Care About Sequence: Why? (1 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

We DO Care About Sequence: Why? (2 of 6)

A. We **leverage sequence** as a **"logic tool"**
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- Sequence is how we know anything at all!
  - What **state** exists
  - How that state **changes/mutates** over ***time***

A

Some **guarantees must exist** regarding **sequence**, or we cannot reason at all.

We DO Care About Sequence: Why? (3 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- **Imperative** Programming
  - Sequence of, "**Do This!**"
- Anatomy and behavior of **algorithmic constructs**
  - **Sequence** (*order*)
  - **Selection** (`if`/`else`, `switch`)
  - **Iteration** (`do`, `while`, `for`, `foreach`)

B "Sequence" is a "**building-block**" (stepping-stone) in **logic training**!

We DO Care About Sequence: Why? (4 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- "**Stepwise-reasoning**" introduces students to programming
- **Is simplest (most atomic)** level of reasoning
- **Is easier** to do "**ordered-walking**" **of myopic cases**, rather than understand the processing "big-picture"

C **Assumptions** related to "sequential reasoning" **become seductive**

We DO Care About Sequence: Why? (5 of 6)

A. We **leverage sequence** as a "**logic tool**"
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it "**adds value**"

- Especially in **large systems** (where we do not know where to get started)
- Especially in **complex systems** (where it is hard to reason about "big-picture" behavior)

D

Sequence is our "**Go-To" tool** when we are "**lost**" or "**stuck**"

We DO Care About Sequence: Why? (6 of 6)

A. We **leverage sequence** as a **"logic tool"**
   - (algorithms)

B. It is **how we were trained**
   - (especially for "higher-level" languages)

C. Can be **easier to understand**

D. We **think** it **"adds value"**

There are **other** ways to think!

Programming paradigms like *"Declarative"* do not rely upon sequence **at all**!

BUT:

- **Many** of these expectations **(assumptions) "do-not-hold"** for **how things actually work** in the compiler/interpreter, and within the CPU
- **Alternative programming paradigms**, and **the C++ Language** *(explicitly!)* do not grant "sequence" this level of reverence

A “Definition-Of-Terms” Problem!

- How does “**Sequence**” relate to “**Algorithm**”?

An **algorithm**
is NOT defined
by the **sequence of statements**
the programmer types.

An **algorithm**
is NOT defined
by the **sequence of statements**
the programmer types.

An **algorithm**
IS defined
by the **sequence of dependencies**
the programmer types.

Corollary: **Sequence** is a **tool** to **manage dependencies** across instructions.

What Defines An "Algorithm"?

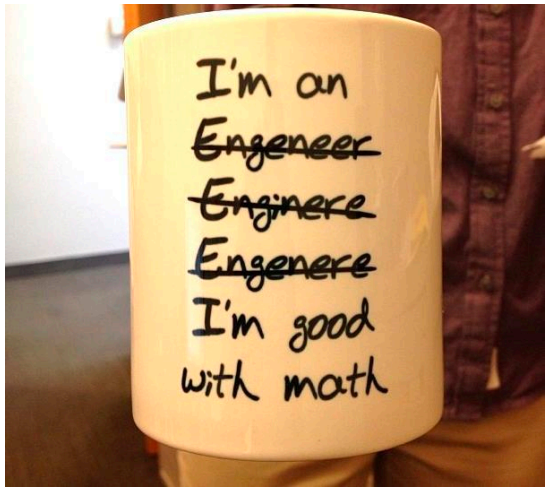
Algorithm
is defined by
logical dependencies
(*not* physical sequence)

- **Example Physical Sequences**:
  - The **compiled program** (instruction order is "fixed" on-disk)
    - *Is "same-program"* if `inline` functions or not
    - *Is "same-program"* after re-compile with different layout
    - *Is "same-program"* compiled with different optimization levels
  - The **loaded program** (instruction order is "fixed" in memory)
    - *Is "same-program"* after address-load randomization
    - *Is "same-program"* after dynamic linking/loading
  - **Self-mutating virus**
    - *Is "same-program"* after (randomizing, mutating) physical instruction sequences

Can change **physical sequence** (logical sequence remains intact)!

Is "Same Program" because algorithm exhibits the same **logical dependencies**!

Job: Software Engineer



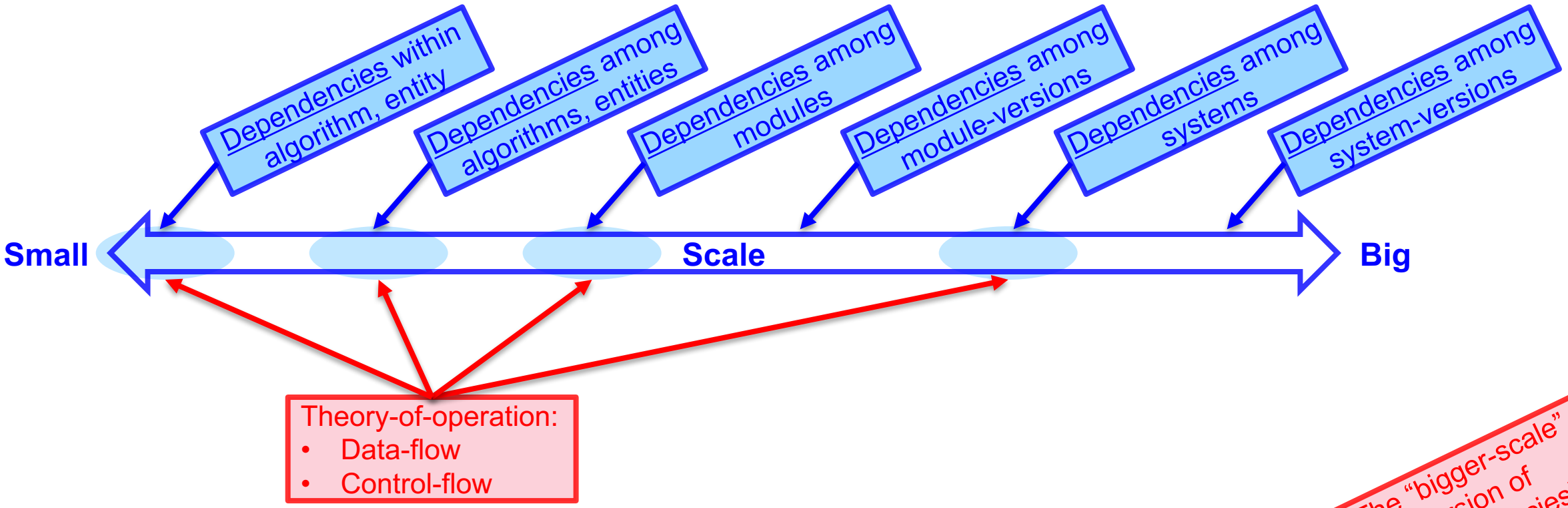
- *Software Engineer*: Our job is **NOT** to **manage lines-of-code**!
  - *Might be your full-time job if you live "Configuration Management" (CM)*
- These are **MECHANICAL operations** to "**keep our workshop clean**":
  - **Add** lines-of-code
  - **Remove** lines-of-code
  - **Move around** lines-of-code (e.g., "refactor")
  - **Reformat** lines-of-code (e.g., "beautify")
  - **Manage** lines-of-code (e.g., "version control")
  - **Build/ship** lines-of-code (e.g., "installer", "CM")

Is NOT software engineering! Is just "part-of-the-job"!

Are based on INTERACTIONS among DEPENDENCIES!

Software Engineer: Our job **IS** to **manage logical dependencies**!
- **Analyze and understand**: Behavior, Capabilities, Alternatives
- **Design and implement**: Algorithms, Features, Functional Changes

Logical Dependencies



- **Logical Sequence** is how we reason about behavior
  - At **"Small" scale**, reason about **dependencies**, **order-of-operations**
  - At **"Big" scale**, reason about **coupling**, **transactions**, **status-and-phases**

The "bigger-scale" version of "Dependencies"

Home | Sims | Stage | Tiles | Trains | Wang

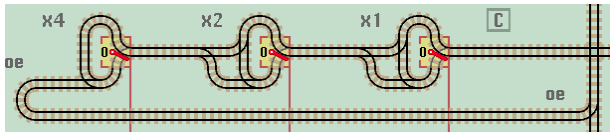
Turing Trains

computational train track layouts

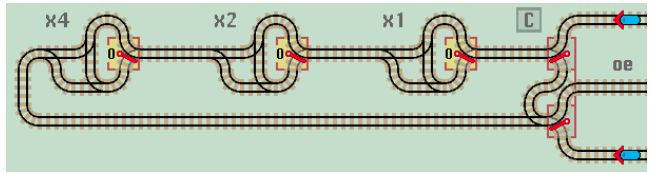
http://www.cr31.co.uk/stagecast/trains/tt2_func_basic.html

Reasoning About Sequence

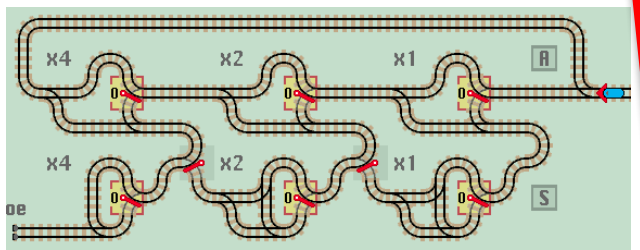
Clear and Error Line



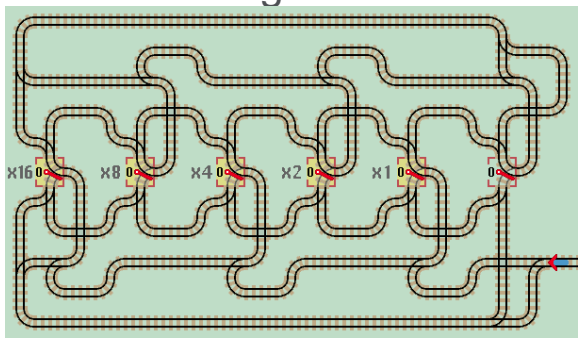
Count and Clear Function



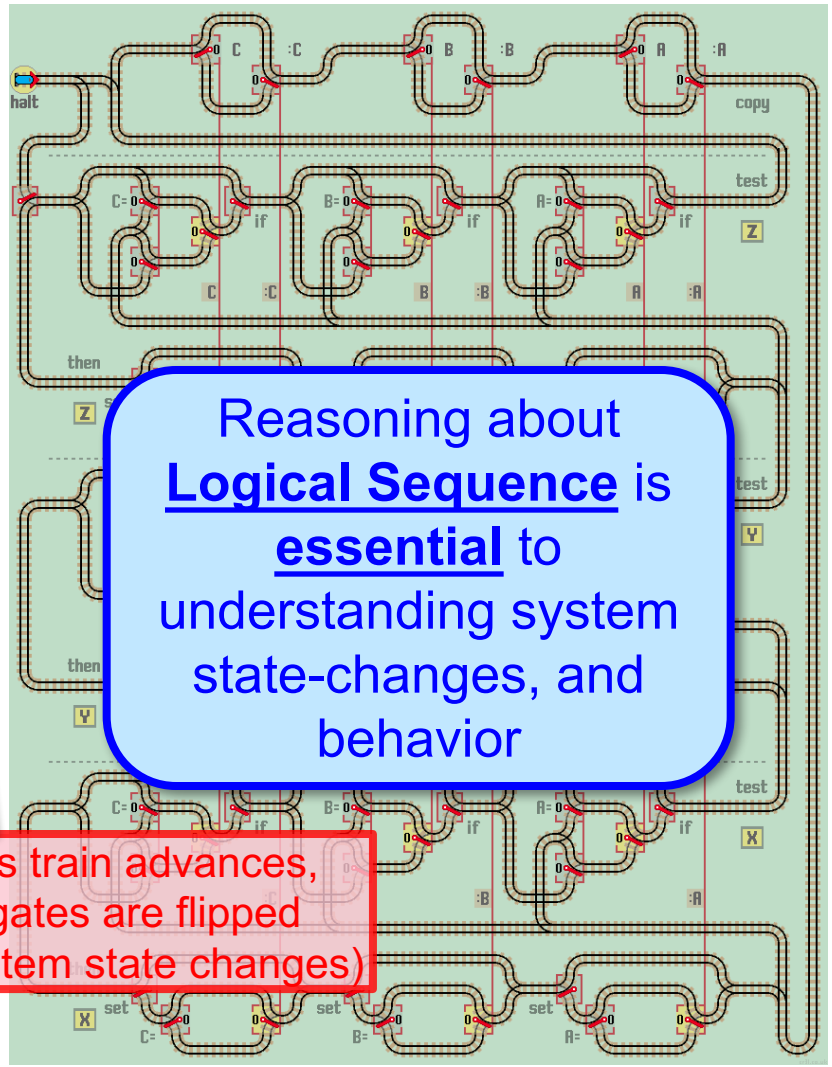
Accumulator Function



Rotate-Right Function



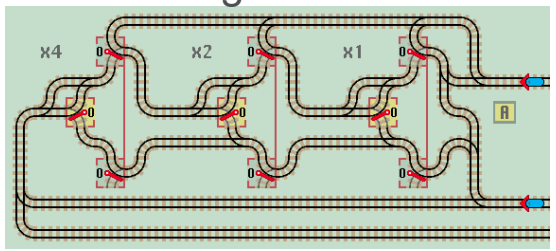
Triangular Number Series Calculator



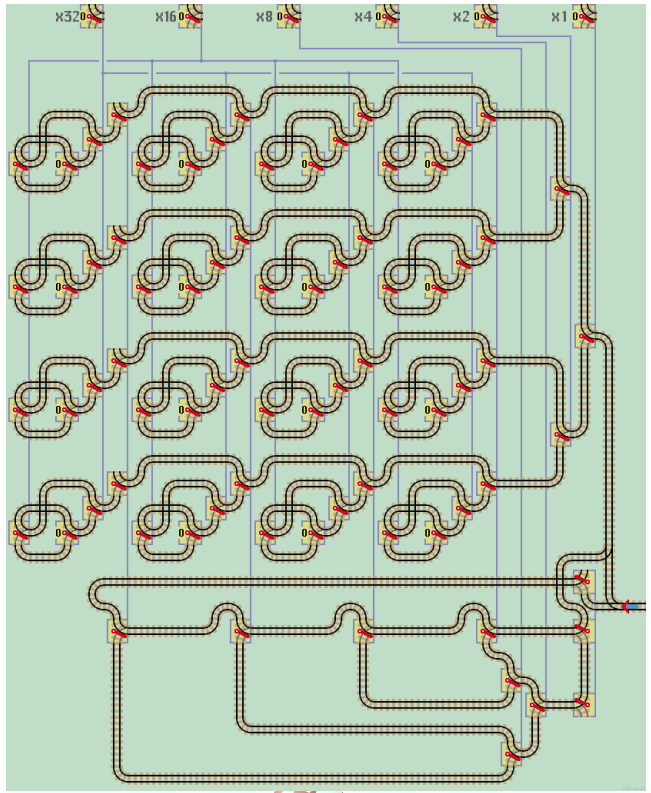
Reasoning about **Logical Sequence** is **essential** to understanding system state-changes, and behavior

As train advances, gates are flipped (system state changes)

Leading Zero Latch



16-Bit Random Access Memory



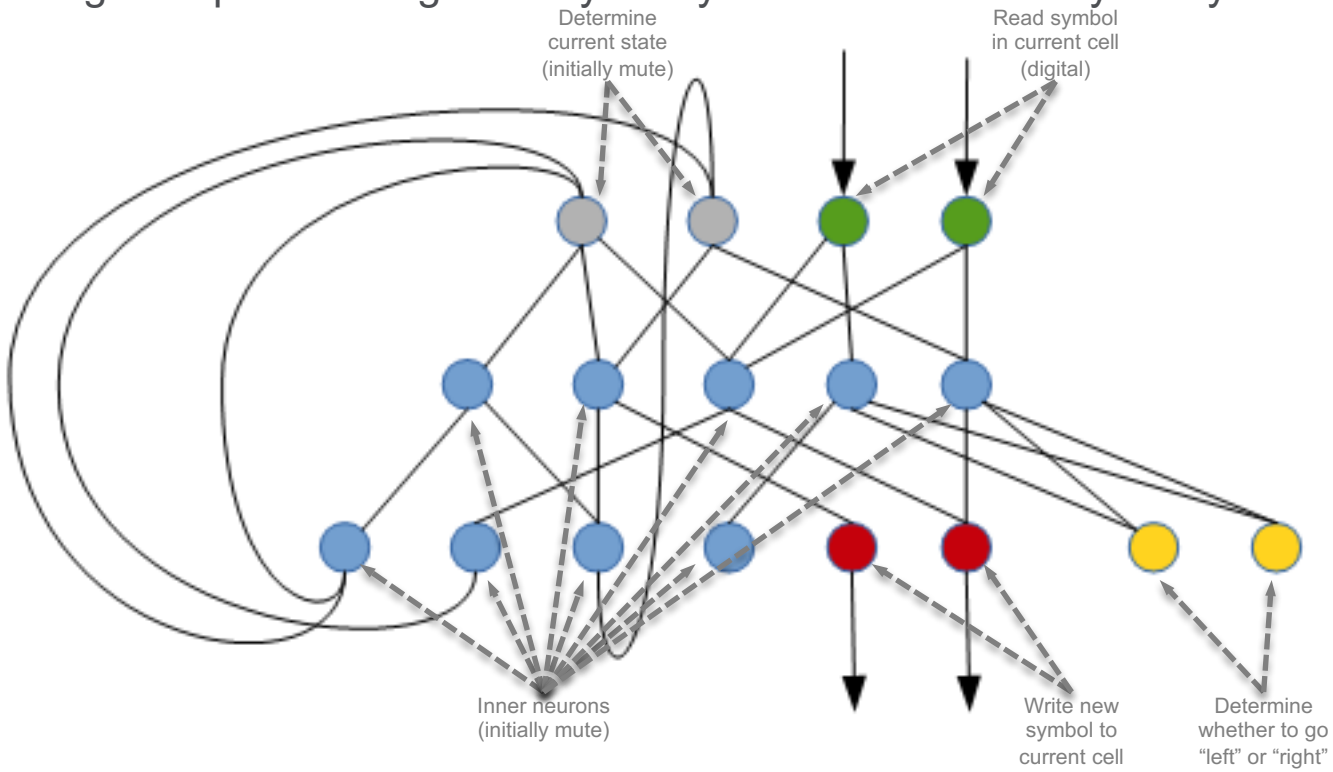
Recurrent Neural Network: *Reasoning About Sequence*

- **State-machine transitions** imply pre/post values based on change-in-state

Reasoning about
logical dependencies,
mutated through
sequential execution,
is required to understand the system

Recurrent Neural Network

Turing-Complete through finitely many neurons with finitely many states



Adapted from Hans Stricker
http://stackoverflow.com/questions/2990277/how-useful-is-turing-completeness-are-neural-nets-turing-complete

Why NOT Care About Sequence?

*So **unnecessary constraints** do not creep into our systems*

Controls Are Constraints

- **Constraints**: We need them! They **define our system**!
  - **Necessary Constraints:** Our system **behaves as expected**
  - **Unnecessary Constraints:** Restrictions that provide **negative-value**
  - **Missing Constraints:** Our **system fails** in some scenarios
- **Controlling** (constraining) for **Sequence**:
  - Want to **constrain** for "**necessary**" sequence
  - Want to **NOT constrain** for "**unnecessary**" sequence

How do we know if controlling a given sequence is "necessary" or "unnecessary"?
Hint: **Dependencies!**

There is **no such thing** as a "**neutral constraint**"
By definition:
Constraints restrict behavior for
desirable (*positive*) or **undesirable** (*negative*) result

Exerting Necessary Control

- Want to **exert NECESSARY control**
  - Specifying **necessary detail is how we design/implement**!
    - …should not be violated!
    - …becomes our design/implementation ASSUMPTIONS!
- Want to Be **SILENT** on **unnecessary detail**
  - Is not "requirement" – we do not care
  - Do not want to restrict OPTIONS
  - Do not want to restrict THINKING
  - Do not want to add unnecessary ASSUMPTIONS

Exerting necessary control is how we **Design and Implement**!
It **establishes behavior**, **expectations**, and allows us the safety to **make valid assumptions** when reasoning about the system!

Avoiding Unnecessary Control

- Want to **avoid UNNECESSARY control**
  - Unnecessary controls create **unnecessary assumptions**
    - ...which are eventually violated
    - ...which restrict your options
    - ...which limit your thinking
  - Unnecessary controls create **unnecessary constraints** (controls *are* constraints!)
    - ...which **disallow efficiencies and technologies** within the compiler and CPU

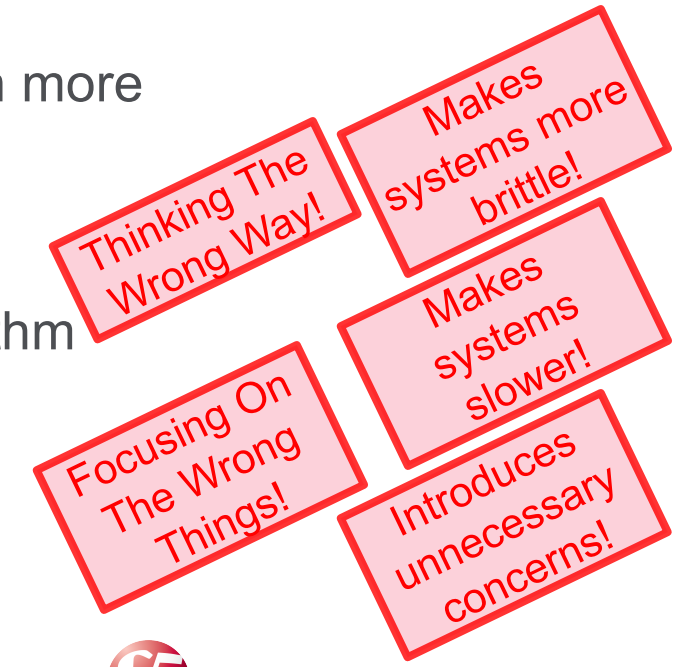
You're not leveraging the compiler and CPU!
(You disallow them from fully doing their jobs!)

Penalties from inflexibility!
(Benefits from flexibility!)

The **More Control You Exert**, the more likely **you hurt yourself and your algorithm**
- …through **misplaced and incorrect assumptions**
  - …that eventually get violated
  - ...and which disallow efficiencies

Why Managing Instruction Order Is Bad

- **Efficiency** Penalties
  - We **refuse to leverage** the **technologies in the CPU**
  - We **disallow optimizations** the compiler provides
  - **Future** system **performance/maintenance** scales **worse**, not better
- **Correctness** Penalties
  - The algorithm is **more likely to be wrong** (more corner cases)
  - We **blind ourselves to alternatives** that scale increasingly better with more resources, larger data sets, and new technologies
- **Effort** Penalties
  - *(usually),* it is **more work to encode more assumptions** in the algorithm
  - *(usually),* it is **more work to hold more assumptions** in our brains



Penalties All Around!

Assumptions: Avoid When Possible

- Most programmers have **too many *(assumptions)***
  - We must **ALWAYS** manage **"some"** assumptions
- You **SHOULD NOT CARE**!
  - about many things, including **instruction order**
- Should: Make **NO ASSUMPTIONS** regarding **"order"**!
  - Should accept that "Instruction Order" is highly ***fickle***, ***erratic***, and seemingly ***capricious***.
    - We should **not want to control** it
    - We should **not want to care** about it
      - *Can design/implement to where we do not care about order!*

If **no** dependencies exist, then **"order" is irrelevant!**

We DO NOT design and implement based on "instruction order". *(We don't care!)*
We **design and implement** based on **dependencies**.

Example Scenario: (Order) Assumptions In Design

Assumptions creeping into our design, into our implementation

“You Fly, I Buy”

...A scenario (with assumptions)

- Example Problem Statement:

“I will pay for the pizza, you go get it.”

- **How many assumptions** will we make?

Algorithm

...or "Design", or "Control-Flow", or "User-Story", or "Implementation", or ...

1. @Me specify, "Supreme w/extra olives"
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

What is the **necessary order**?

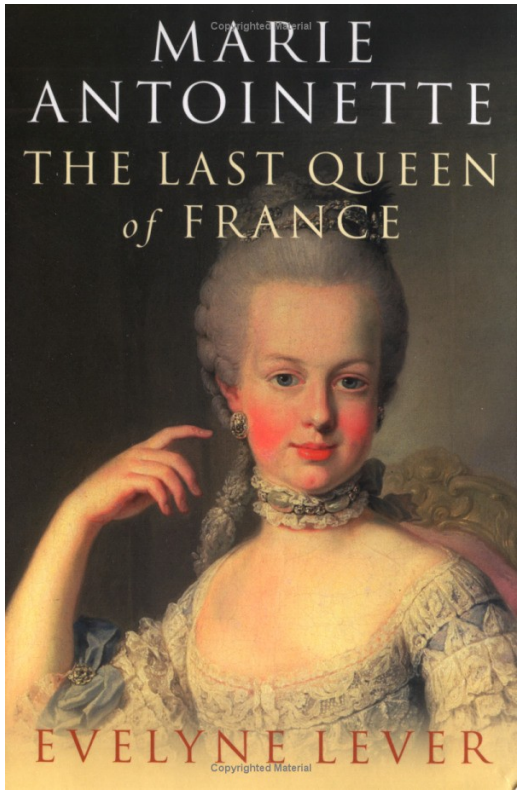
For which should we **NOT CARE** regarding order?

Order That Matters

...Necessary constraints (warranted coupling establishing order)...

1. @Me specify, "Supreme w/extra olives"
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

Cannot Eat Before Have!



What is the **necessary order**?

For which should we **NOT CARE** regarding order?

On “1. @Me Specify…”

…Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

Can give you money first!

Can phone you with order after you leave!

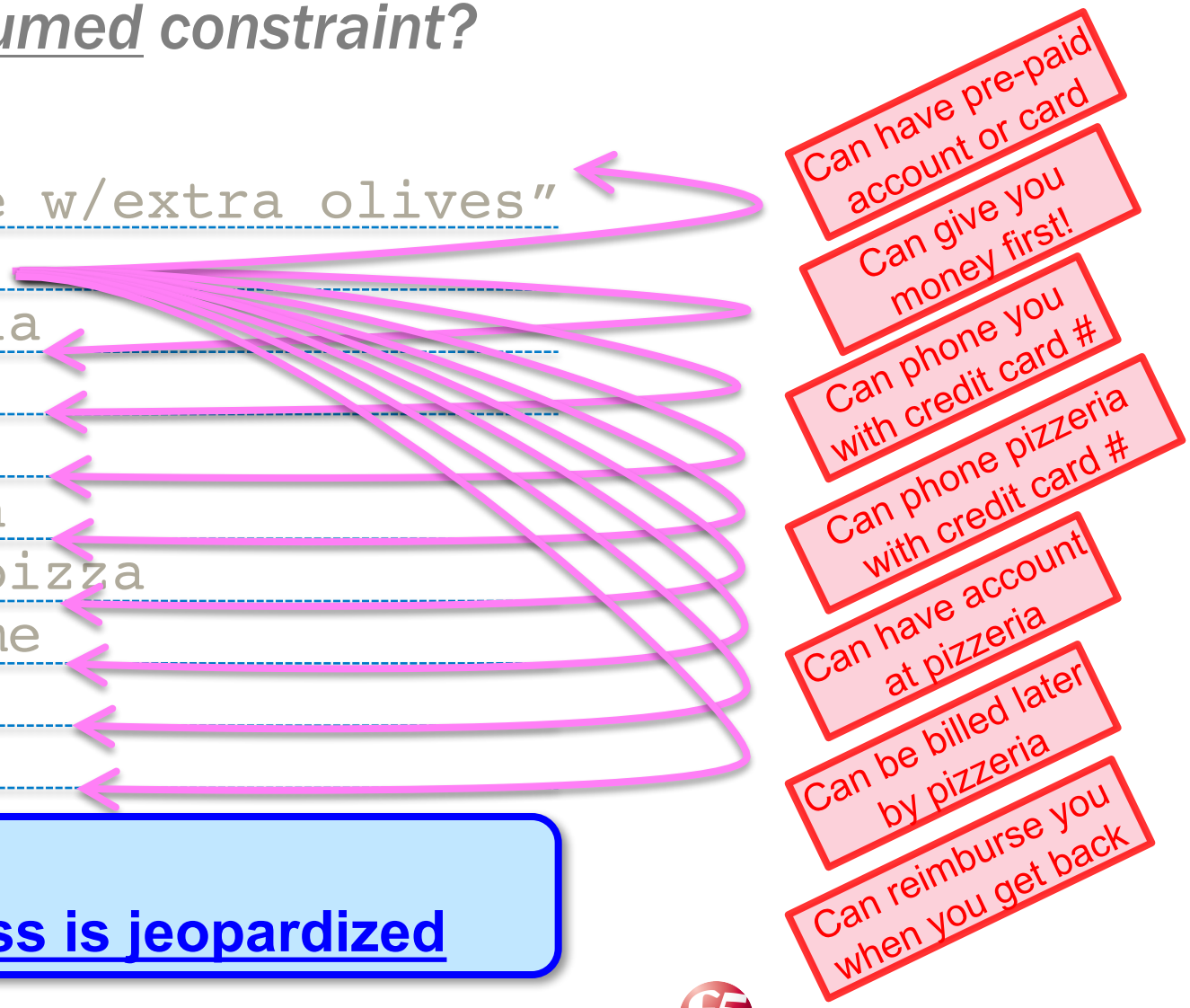
Why care about sequence?

- ONLY if you **get the wrong pizza**

On “2. @Me Give You Money”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



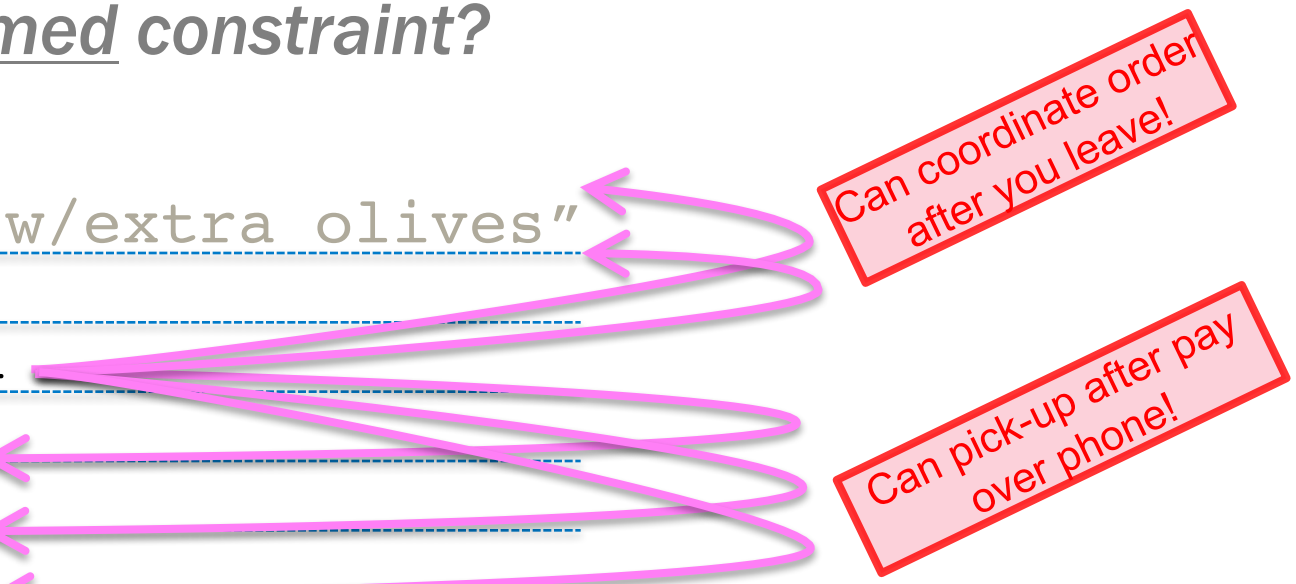
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “3. @You Drive To Pizzeria”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



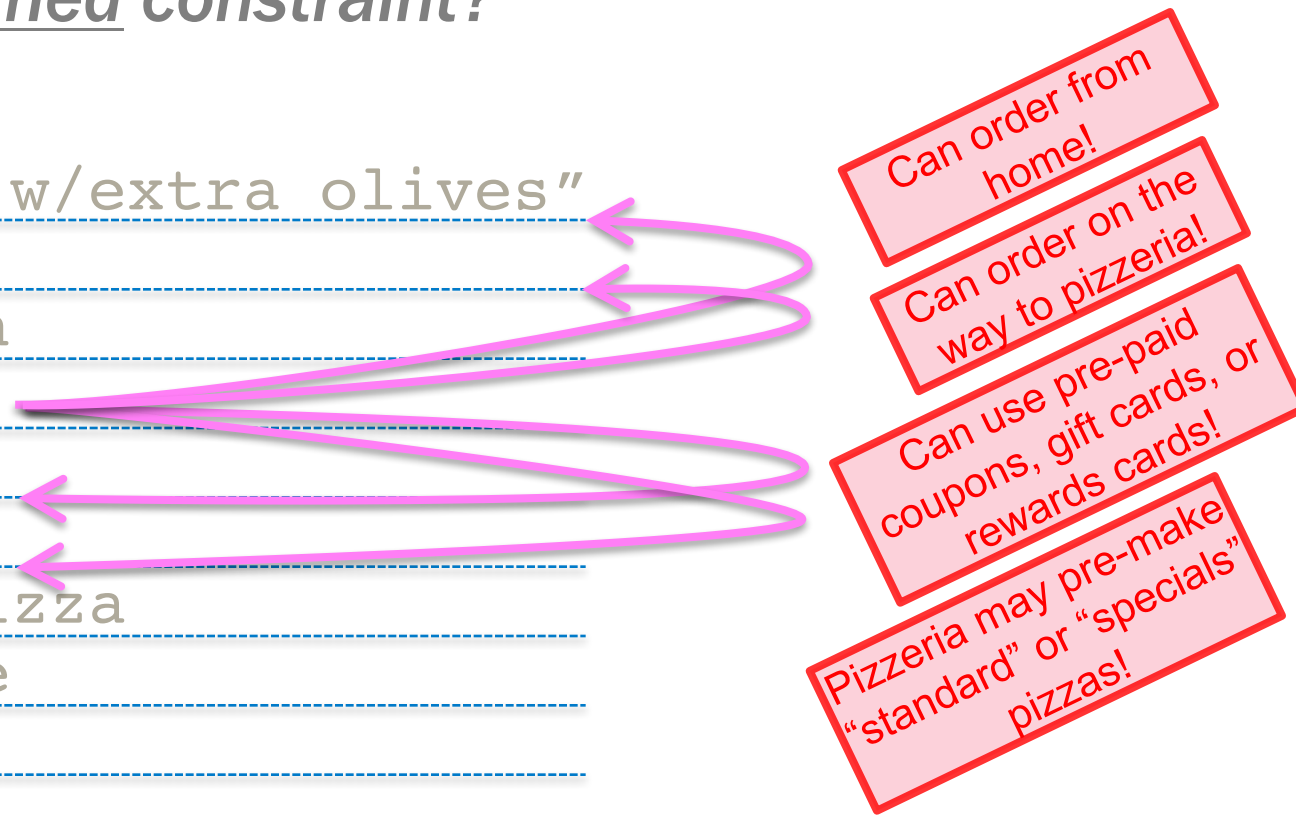
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “4. @You Order Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



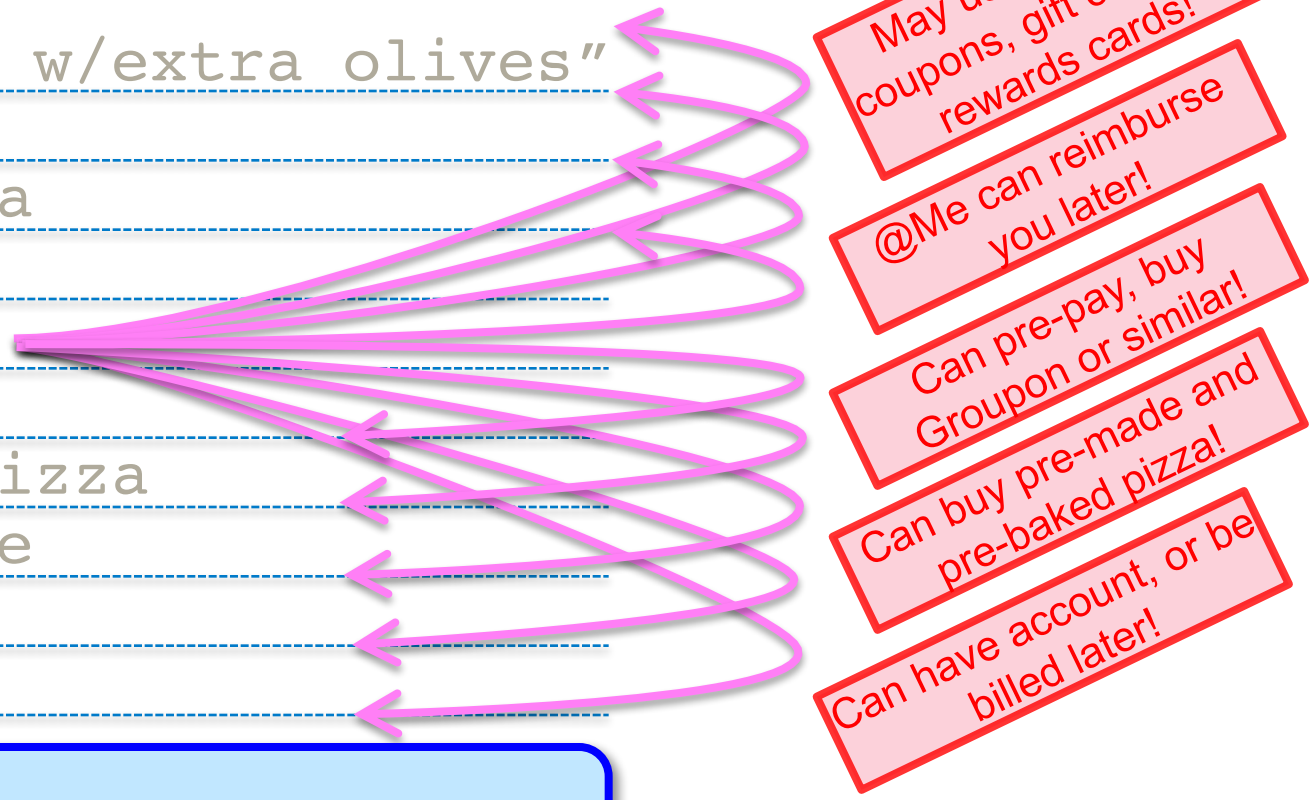
Why care about sequence?

- ONLY if you **get the wrong pizza**

On “5. @You Pay For Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



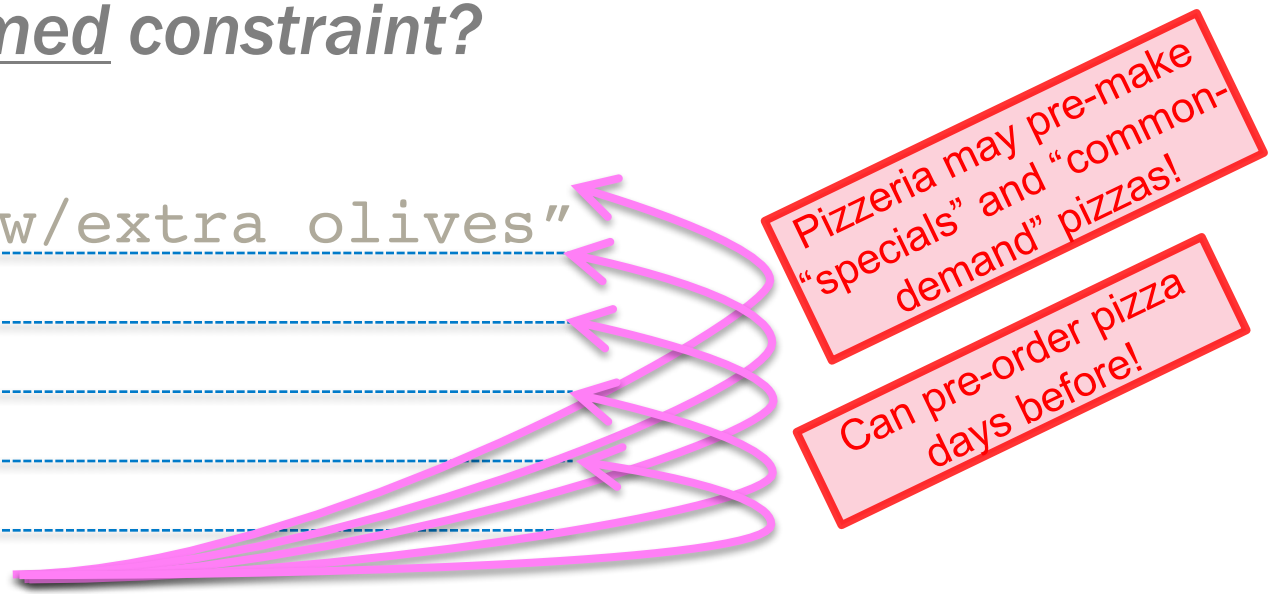
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “6. @Pizzeria Makes Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. **@Pizzeria makes pizza**
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



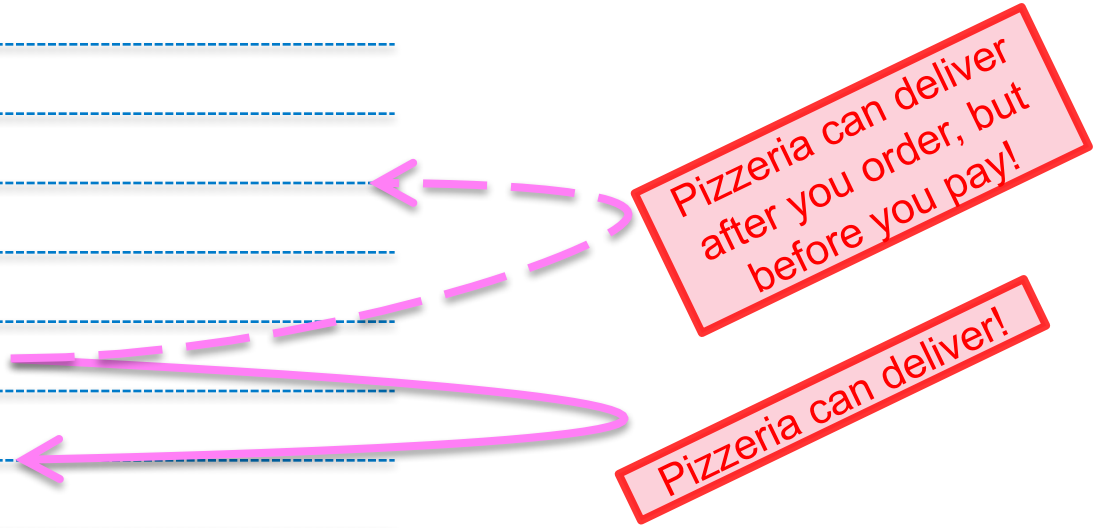
Why care about sequence?

- ONLY if you **get the wrong pizza**

On “7. @Pizzeria Gives You Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. **@Pizzeria gives you pizza**
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “8. @You Bring Pizza To Me”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

Pizzeria can deliver! (Remove instruction?)

Pizzeria can deliver! (Remove instruction?)

Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “9. @Me Has Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza

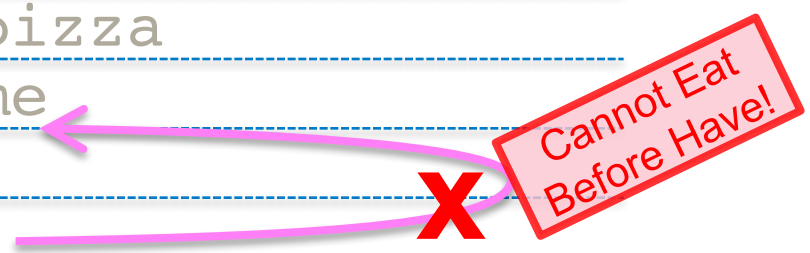
Why care about sequence?

- ONLY if **transaction success is jeopardized**

On “10. @Me Eats Pizza”

...Necessary constraint, or Assumed constraint?

1. @Me specify, “Supreme w/extra olives”
2. @Me give you money
3. @You drive to pizzeria
4. @You order pizza
5. @You pay for pizza
6. @Pizzeria makes pizza
7. @Pizzeria gives you pizza
8. @You bring pizza to me
9. @Me has pizza
10. @Me eats pizza



Why care about sequence?

- ONLY if you **get the wrong pizza**
- ONLY if **transaction success is jeopardized**

Scenario Review: We DID Consider...

- We considered the **algorithm**
  - The "**Goal**"
  - **Sequence-of-steps for how** the "Goal" is achieved
  - **Dependencies among instructions** within algorithm
    - Including **flexibility** for "**Instruction Re-Ordering**"

We focused on ONE algorithm!
There are an **infinite number** of **alternative approaches** (designs) to this problem!

- **In Considering "Dependencies"** among instructions:
  - Identified **corner-cases forced** us to **define-terms** regarding:
    - The **meaning of** specific "**state**"
    - The **meaning of** specific "**operations**" (or "activities")
  - "**What-if" logic** and scenarios **forced** us to **identify** "required" **instruction-order**

Algorithm requires:
- What "**state**"?
- What "**operations**" upon state?
- What "**dependencies**" among state-changes?

Scenario Review: We DID NOT (YET) Consider...

What About…

System "Tuning" might be factory-configured or user-configured

1. **Preferences** *(Optimization Bias)*
   - **How much** are we **willing to relax** some things **to get more** of another thing about which we care more?

(*example*), User-Latency requirements might be sacrificed in some "special circumstances"; Compatibility requirements might be sacrificed for some versions, or configurations

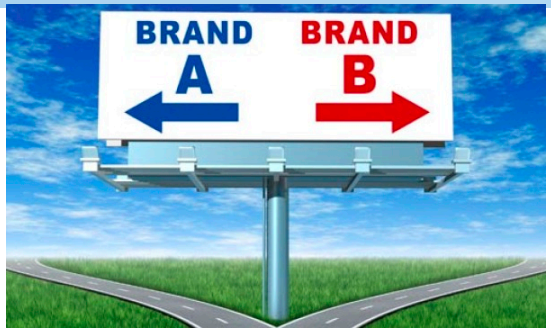
2. **Constraints**
   - **What "limits"** are we **willing to compromise** at some level?
     - (i.e., exploration of ***"soft-limits"*** and ***"hard-limits"***)

System is validated and controlled for characteristics that the user does not care about

3. **Over-Specifications**
   - What "**controls/limits**" are **unnecessary**? (We **mistakenly *ASSUMED***?)
     - What **ASSUMPTIONS restrict** our system from achieving more optimal/desired levels-of-performance?

What About "Preferences"?

For what do we optimize?



Can "Rank Order" to perform "Optimization Bias"!

Optimizations can influence *"selected-instruction-order"*!

- Prefer ***"Hot"***
  - Select pizzerias closer-to-home
  - Order after you get there (not ahead-of-time)
  - Select "Pick-Up", not Delivery
- Prefer ***"Good"***
  - Select pizzeria across town with better pizza
- Prefer ***"Cheap"***
  - Select low-cost pizzeria
  - Select pizzeria with "specials"
  - Select pizzeria for which we have coupons
  - Select pizzeria for which we have "rewards-card"
  - Select "Pick-Up", not "Delivery" *(so no $ for tip!)*
- Prefer ***"Fast"***
  - Select pizzeria closer-to-home
  - Select pizzeria that "pre-makes" pizza
  - Select pizzeria with faster production pipeline
- Prefer ***"support-specific-business"***
  - Select pizzeria recently started by your cousin Vinni
  - Select pizzeria employing the person with whom you are trying to get a date
- Prefer …?

What About “Constraints”?

How are our choices limited?

The Big Bang Theory (2007 -)

Constraints
can influence
“possible-instruction-order”!

- **Vehicle** available
  - Limits pizzeria selection based on distance, motivates Delivery not Pick-Up
- **Money** available
  - Limits pizzeria selection based on price-range, current-deals, on-hand coupons, frequent-buyer punch-card, Pick-Up not Delivery
- **When:** Day-of-week, Time-of-day, “The Big Game Day”
  - Limits pizzeria based on who is open, customer-load (e.g., long wait), etc.
- **Dietary restrictions**, preferences
  - Limits pizzerias to those with vegetarian/vegan, gluten-free, etc., or those offering the pizza type desired

Biggest Assumption: Over-Specification (1 of 2)

"Unwarranted Constraints"

- Scenario Purpose:
  - **@Me pay, but @Me not leave the house**
- Over-specification:
  - **@You will go get it**
- Actual Specification:
  - **@Me not leaving the house**
- Impact:
  - Should "Delivery" be considered (in addition to "Pick-Up")?
  - Should "Take-and-Bake" be considered?
  - Should we "Pre-Order" before The Big Day?
  - Should we have frozen-pizza "on-hand" for The Big Day?
  - Should we make pizza at home (from box, from scratch?)
  - Should friend just bring pizza when coming over?
  - Should you take a job at a pizzeria for free pizza?
  - Should you make friends with "Delivery Guy" who can get you all the free pizza you want?
- Other Over-specifications In This Scenario
  - @Me specify vs. @You specify pizza-type
  - @Me order vs. @You order
  - @Me pay, vs. @Me reimburse @You later (or @Me billed later by @Pizzeria)
  - Take&Bake pizza (provides flexibility regarding constraints "time", "hot")
  - Plan Ahead (have frozen pizza "on-hand")
  - …etc.

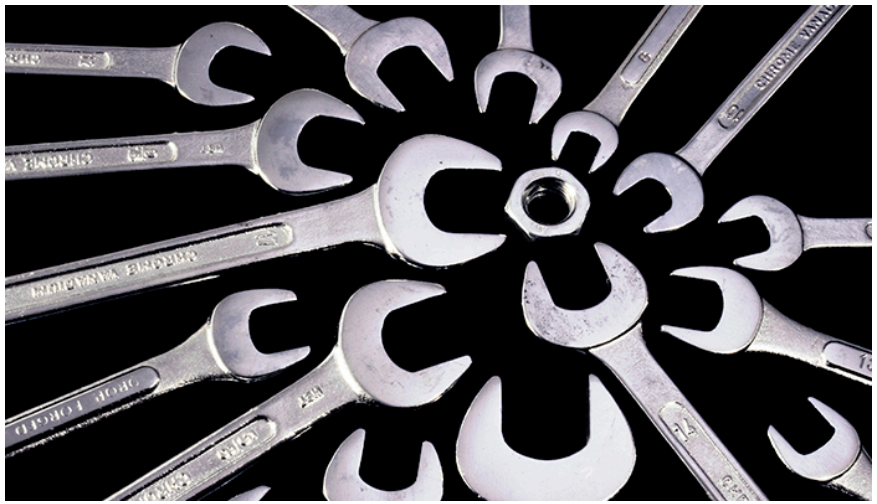
Any assumption unrelated to purpose is not a "real" specification/constraint!

Usually "YES", something else is worth considering!

Over-specifications are **identified** by (*continually*) re-visiting **"First Principles"** (i.e., the system's purpose)

Biggest Assumption: Over-Specification (2 of 2)

"Unwarranted Constraints"



(*often*),
ENORMOUS Benefits
from relaxing constraints!

Over-Specifications

- **Limit options**
  - For **no benefit**
  - For **negative benefit**
- **Confuse/Hide** the actual **constraints**!
  - **Harder to discover** elegant patterns
  - **Harder to reason** about (understand) behavior
- **Slow** our systems
  - by adding **unnecessary constraints**
- **Break** our systems
  - when unnecessary constraints are (eventually) violated

Over-Specification:
Erroneous constraints
should **always** be
identified, and removed!

- **Real constraints** – may be relaxed in some scenarios *(maybe)*
- **Erroneous constraints** – always provide *negative* value

Summary: Influences On Instruction Order (1 of 2)

- Instruction order influenced by:

1. **Requirements**: Sequence demanded through correct algorithmic execution
2. **Preferences**: Optimization biases expressing preferred tradeoffs
3. **Constraints**: Limitations based on what is possible
4. **Translation**: Communication friction where concepts are lost or changed during conversion
   - **REALITY** Hits!
   - Must translate to execute!

The same "System Requirements" can be expressed through different algorithms (each with "algorithm requirements")

May be compile-time, or runtime!

Watch for "over-specifications" ("Phantom Constraints")

Translation, and opportunities for misunderstandings ALWAYS EXIST!

OHZ NOZ! We still need to go through the compiler!

(Hidden) System Failure Scenarios:
Assuming "universal *concept* and *execution* agreement" among all parties.

Summary: Influences On Instruction Order (2 of 2)

- Instruction order influenced by:

1. **Requirements**: Sequence demanded through correct algorithmic execution
2. **Preferences**: Optimization biases expressing preferred tradeoffs
3. **Constraints**: Limitations based on what is possible
4. **Translation**: Communication friction where concepts are lost or changed during conversion

- **REALITY** Hits!
  - Must translate to execute!

Software Developer designs!

Compiler optimizations (speed/size/cross-platform, instruction set compatibility), sometimes by programmer design/algorithm, by system configuration

Mandated CPU cores, threads, execution units, or by system configuration

Many-stage Compiler translation to object code, CPU executes (pipelined) instructions through the CPU cache

Imperative vs. Sequential Devices

"Do This" vs. *Control Structures (Necessary Order)*

Programming Paradigms

C++ is Multi-Paradigm Language! Can do all of them!

- Adapted from: https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms
  - **Imperative**: Statements directly change program state *(e.g., "Do This, then Do That")*
  - **Structured**: Style of Imperative with more logical structure
  - **Procedural**: Modular programming (procedure call), derived from Structured
  - **Functional**: Computation is evaluation of functions, avoiding state and mutable data
  - **Event-Driven, Time-Driven**: Program flow determined by events
  - **Object-Oriented**: Datafields treated as objects manipulated through methods
  - **Declarative**: Defines computation logic without defining control flow
  - **Automata-Based**: Programs are a model of a finite-state-machine or formal automata

Derived from

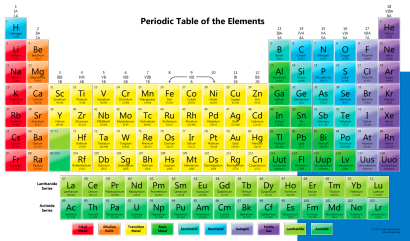
Imperative Devices are controlled through **Imperative** programming

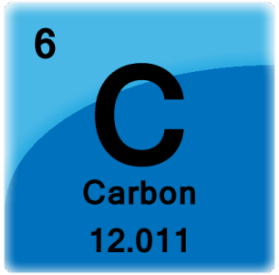
Instruction Order Is Crucial!

Sequential Devices are controlled through **Structured** or **Procedural** programming

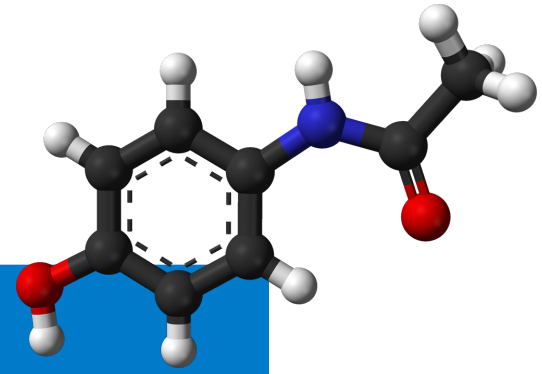
Instruction Dependencies Is Crucial!

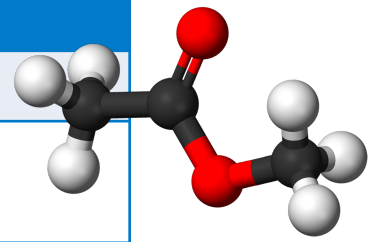
Contrast Imperative vs. Procedural

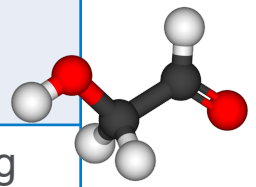


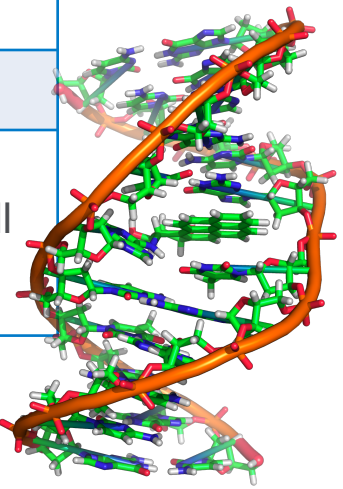


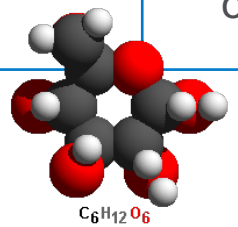
	Imperative: "Do This"	Procedural: Control Structures
Building Block	• **Instruction**	• **Control Structure**
Sequence Is Established By	• (Instruction) **Order**	• **Logical Dependencies**
Practical Application	• Interesting to explain **atomic-level reasoning**	• Interesting to explain **all life** and **processes we observe**
Metaphor	• **Atom** (association of indivisible smallest parts)	• **Molecule** (association of interacting compounds)
Explains	• **Table Of Elements**	• **All compounds that actually exist**
Basis For	• **Nuclear Chemistry** (including the most expensive way to "boil water" ever invented)	• **Electron Chemistry** (including all human-observable processes and all organic life)

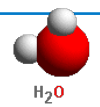


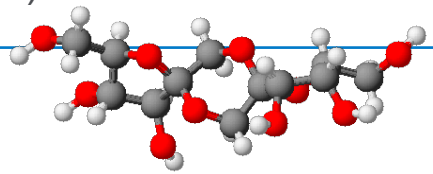






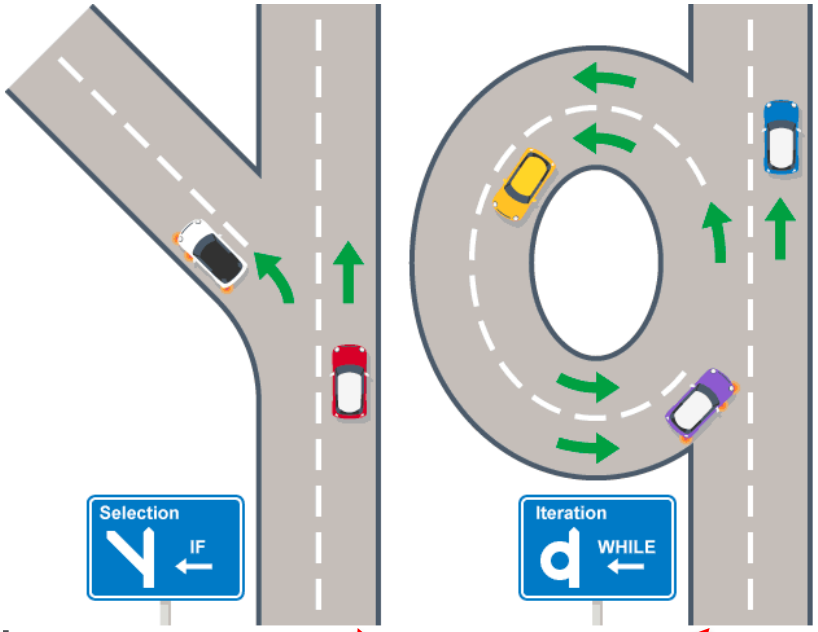






Sequence Of What?

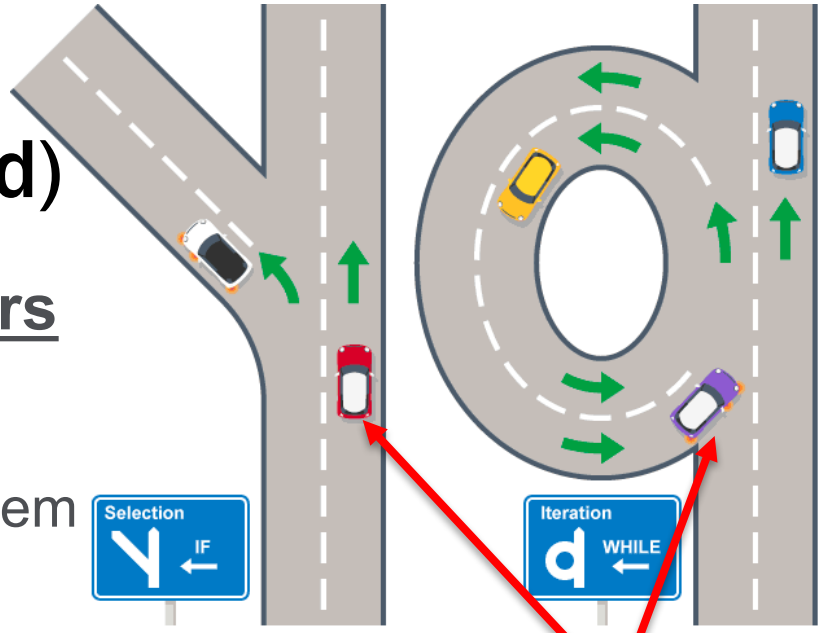
- Sequence of what?
  - ~~Sequence of instructions~~
  - ~~Sequence of data~~
  - Sequence of **dependencies**
- When we speak of, “Sequence”, are we talking about the “road” or the “cars-on-the-road”?
  - ~~Are we discussing the structure of our lines-of-code?~~
  - Are we discussing the **data-flow and logic-flow defined by the dependencies established** through our lines of code?
- Is “Sequence” the **road** (our lines-of-code), or the **cars** (that actually move/flow through our system)?



We **design and implement** our systems by defining and building the **roads** on which data and instructions will flow!

Sequence Is The Cars (Not The Road)

- We **Design the Road**, to (indirectly) **Influence the Cars**
- **The Road**: **Program Structure, Control Flow**
  - We write lines-of-code (we “build-the-road") to craft our system
  - Lines-of-code is NOT the “sequence”!
- **The Cars**: **Data-Flow, Instruction Sequence**
  - Data-and-Instructions “move/flow” to run our system (the cars)
  - Lines-of-code (the “road”) is an INDIRECT INFLUENCE on “sequence”!



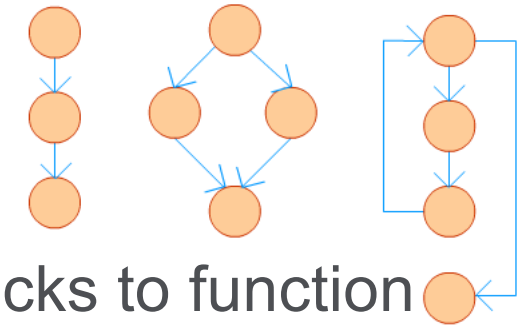
Sequence is the “Cars”, which we (indirectly) control by defining the “Road”!

Can define any road you want!

Software Engineering establishes the structure (the roads) **to INDIRECTLY control data flow, logic flow** (the cars) that represent the processing we REALLY care about.

Algorithms Are Formed

- **Algorithms** are formed **entirely from building blocks**:
  - **Sequence** (ordered series of steps)
  - **Selection** (branched path, some not taken)
  - **Iteration** (repeat some instructions)



- **The "mechanisms"** that enable the building blocks to function (conditional values, counters) **establish the dependencies**
- We **ONLY care** about "**Sequence**"!
  - **Selection and Iteration** are merely convenient (logical utility) constructs to **assist** us **in defining sequence** (Turing-Complete algorithms)
  - Hard to otherwise **define an "infinite-loop"** sequence, or ***"select-among-option-set"*** sequence

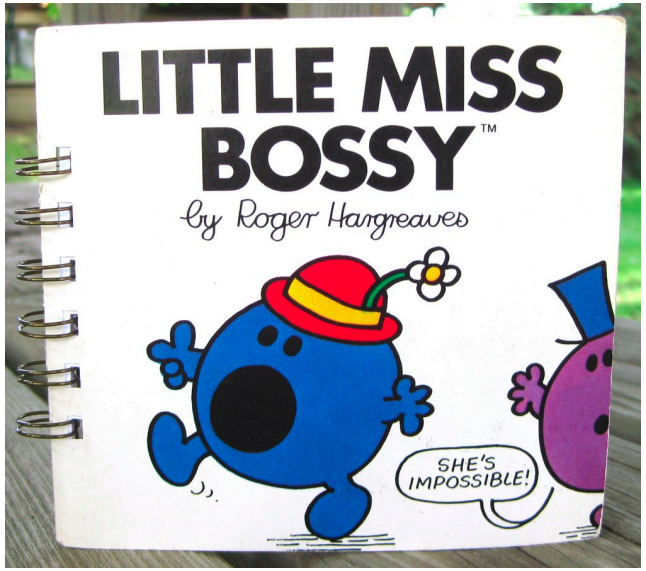
Algorithmic constructs provide efficient syntax to express sequence!

Sequence is defined by dependencies!

"Selection" and "Iteration" are merely constructs to assist in expressing (describing/defining) sequence!

Sequence is otherwise termed, "**Sequence of Dependencies**"
(NOT "sequence-of-data" nor "sequence-of-instructions")

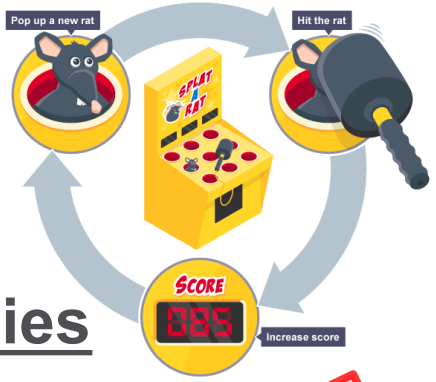
Review: Imperative



- "**Imperative**" means we specify "**Do This!**", and it is done
  - **Minimal** (or no) **latitude**
  - **High control** (restriction) regarding behavior
  - **Very repeatable** (assuming no error, and no external state-changes)
  - *(Usually),* Highly **Inefficient**
    - You must "wait" until instruction is "done" before issuing next instruction
- Example:
  - Your **well-trained dog** is an imperative device
- Interesting Applications:
  - **Real-Time Operating Systems (RTOS)** demand exact control of each instruction cycle, and each resource allocation

Today's CPUs don't work like this! **Superscalar** CPUs use **"pipelines"!**

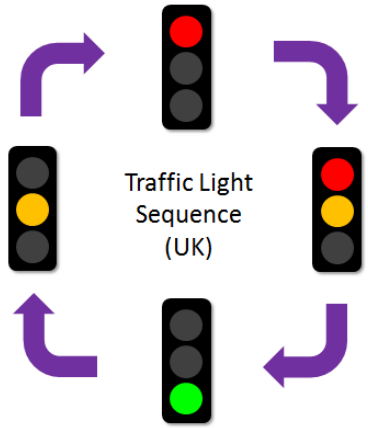
Review: Sequential



- "**Sequential**" assembles **control-structures based on dependencies**
  - Can **reason about behavior** (order is "defined" by logical dependencies)
  - Can **reason about mutations** (e.g., "state-machine")
  - Can be **Turing-complete** (e.g., "Turing Trains" is Turing Complete)
    - Turing Complete: System is able to perform any calculation, given enough resources. *(Non-Turing-Complete systems are unable to handle a specific set of calculations, even with enough resources.)*
    - Turing Complete means algorithm can be expressed (whether-or-not it is performed)

We rely upon logical control structures!

- Example:
  - **Airplane "Auto-Pilot"** is a sequential device *(tends toward stable state by managing internal and external dependencies through well-defined algorithm(s))*
- Interesting Applications:
  - **Most real-world** Software Engineering **systems**, **system-integrations**



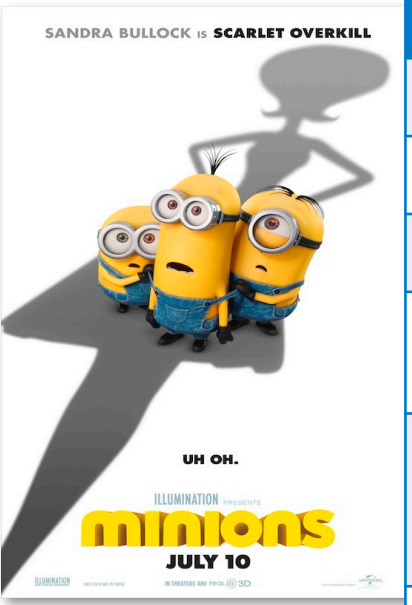
Imperative vs. Sequential Device

Note: Today's Distributed Web architectures scale through "*Agents*", not "*Minions*"

Dave

Agent Smith

Imperative: "Do This"	Sequential: Logical Steps
• Is "**Task-based**"	• Can be "**Goal-based**"
• Establishes "**next-step**" (only)	• Defines **logical procedure** (algorithm)
• **Oblivious** to logical dependencies	• **Relies upon logical dependencies**
• **Is Limited**: Requires constant (frequent) supervision and control	• **Can be used** as **organizational structure** (to build/scale systems)
• **Hard to work with**: Must specify many seemingly irrelevant details to address unexpected/surprising behavior	• **Easy to work with**: Operates with latitude within established constraints
• **CANNOT** scale	• **CAN** scale

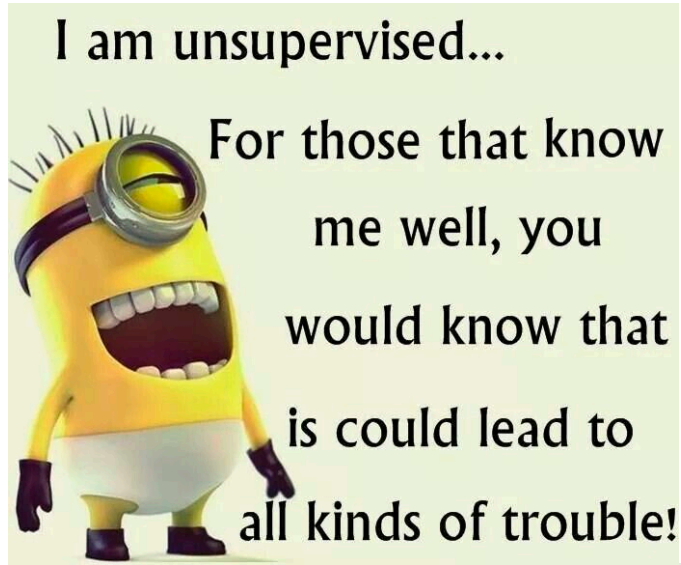


Minions (2015)

The Matrix Reloaded (2003)

What It's Like To Work With Imperative Devices

- You must:
  - **Queue all future tasks** *(the device will not)*
  - **Perform all scheduling** *(the device will not)*
  - **Identify all dependencies** *(the device will not)*
  - **Manage all dependencies** *(the device will not)*
  - **Identify opportunities** for efficiencies *(the device will not)*
  - **Respond to unexpected** external or system events *(the device will not)*
  - **Concurrently interface** with other components/systems *(the device will not)*
  - **Continually establish further constraints** when the device exhibits surprising behavior



Just shoot me now.

Exceptions reset to "baseline" state. There is no baseline state (chaos).

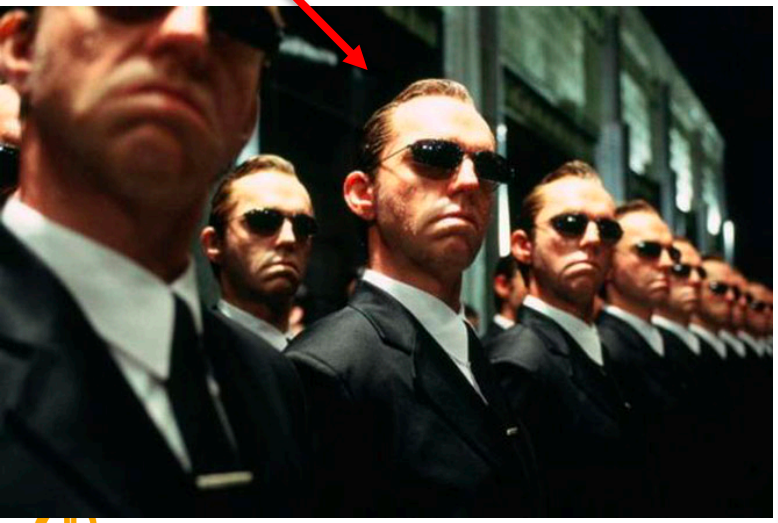
No context exists to throw exception, because all behavior is "custom one-off"

What It's Like To Work With Sequential Devices

- You must:
  - **Establish algorithm** based on **logical dependencies**
  - **Establish objectives/priorities** to guide device execution
  - **Establish constraints** for device behavior
  - **Establish rules** for **concurrent interfacing** with external components/systems
  - *That's it!*

Stable monitoring, idle mode

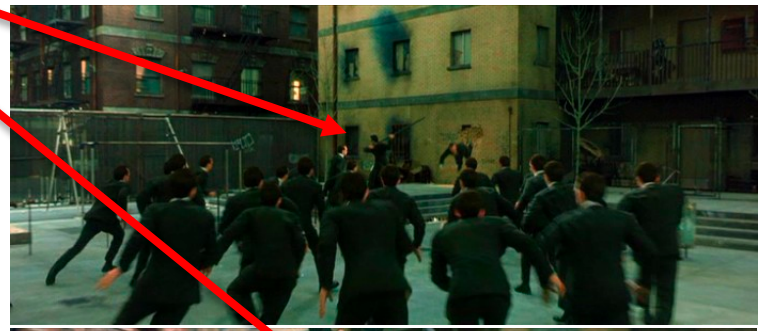
Self-organizing



Variable load-response based on internal metrics

Goal-oriented

Exception is thrown



Taking over the world is far easier.

Pinky & the Brain (1993-1999)

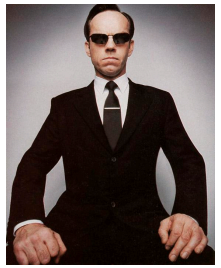
Contrasting Imperative vs. Sequential Devices

- **Imperative** devices:
  - Are "**Simple**": **Easy to create**, and to **understand**
  - At Scale: Approximate **Chaos**
    - Logical dependencies **not respected**
    - **Must tightly supervise interactions** among components that do not respect (*logical*) dependencies
  - **Failure to Supervise: Chaos** *(possibly death – yours or theirs)*

- **Sequential** devices:
  - Are "**Higher-Order**" devices **bounded by logical dependencies** (*algorithm*)
  - May be **"simple"**, **or "rich"** in behavior
  - **Scale well**
  - **Failure to Supervise: Predictable Results**, can tend towards "order"

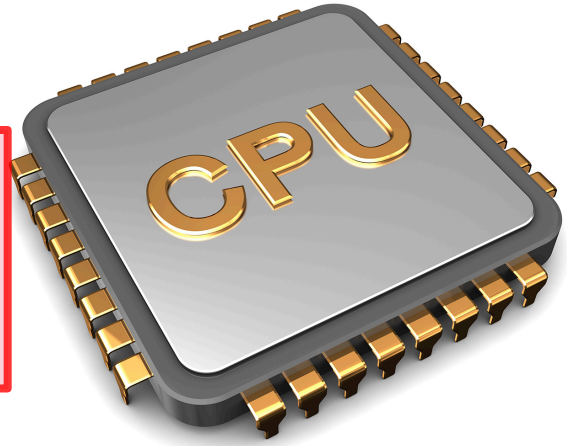
Goal: Is **easier to scale systems with Agents** that follow (established) procedure, versus Minions that must be constantly supervised. (*Chaos vs. Predictable Results*)

Minions (*imperative*) might be fun for entertainment value, and because they are unlikely to threaten you with collective bargaining; but **if you want to get work done** (**efficiently** and **correctly**), and **at scale**, then use Agents (*sequential*).



The CPU Is A Sequential Device

The Modern Super-Scalar Multi-Core CPU Is *Awesome*.



- Very rich **internal technologies**, self-managed
  - Multi-level **CPU cache** (loading, eviction, coherency)
  - **Superscalar CPU Instruction Pipeline** (many instructions started/retired per cycle)
  - **Speculative Execution** (instructions "eagerly" executed, might be aborted, re-tried, or results disregarded)
  - **Rich Microcode** (firmware updates, other optimizations)
  - …and *much more*!

CPU is very rich **Sequential Device**! (Not Imperative!)

Today's CPU will **trace internal** and **external dependencies** to

- parallelize execution, pre-fetch data, and speculatively execute instructions,
- while managing cache coherency *(dependencies!)* across threads, processes, and cores,
- while managing internal metrics for data access patterns and cache hits,
- to constantly optimize *(at runtime!)* based on a given execution sequence, data set, or runtime-observed system load characteristics.

Physical vs. Logical Sequences

What *actually* happened, vs. what you *assumed* happened

Physical vs. Logical Sequence

- **Exactly TWO** kinds of sequence exist:

1 **Physical Sequence** defined by **order** in which instructions "occur"

2 **Logical Sequence** defined by **logical dependencies** in an algorithm

- *"Occur"*
  - Are **specified** in source file?
  - Are **serialized** in object code?
  - Are **presented** through the Program Counter?
  - Are "**started**" (in the CPU)?
  - Are "**completed**" (in the CPU)?
  - **Results** are **propagated/written**?

Physical Sequence is ALWAYS ambiguous!

- *"Logical dependencies"*
  - Are **EXPLICIT** based on:
    - "**Well-Defined Behavior**"
      - ...those logic (control) structures for which the programming language ***EXERTS CONTROL***

Logical Sequence is ALWAYS Well-Defined!

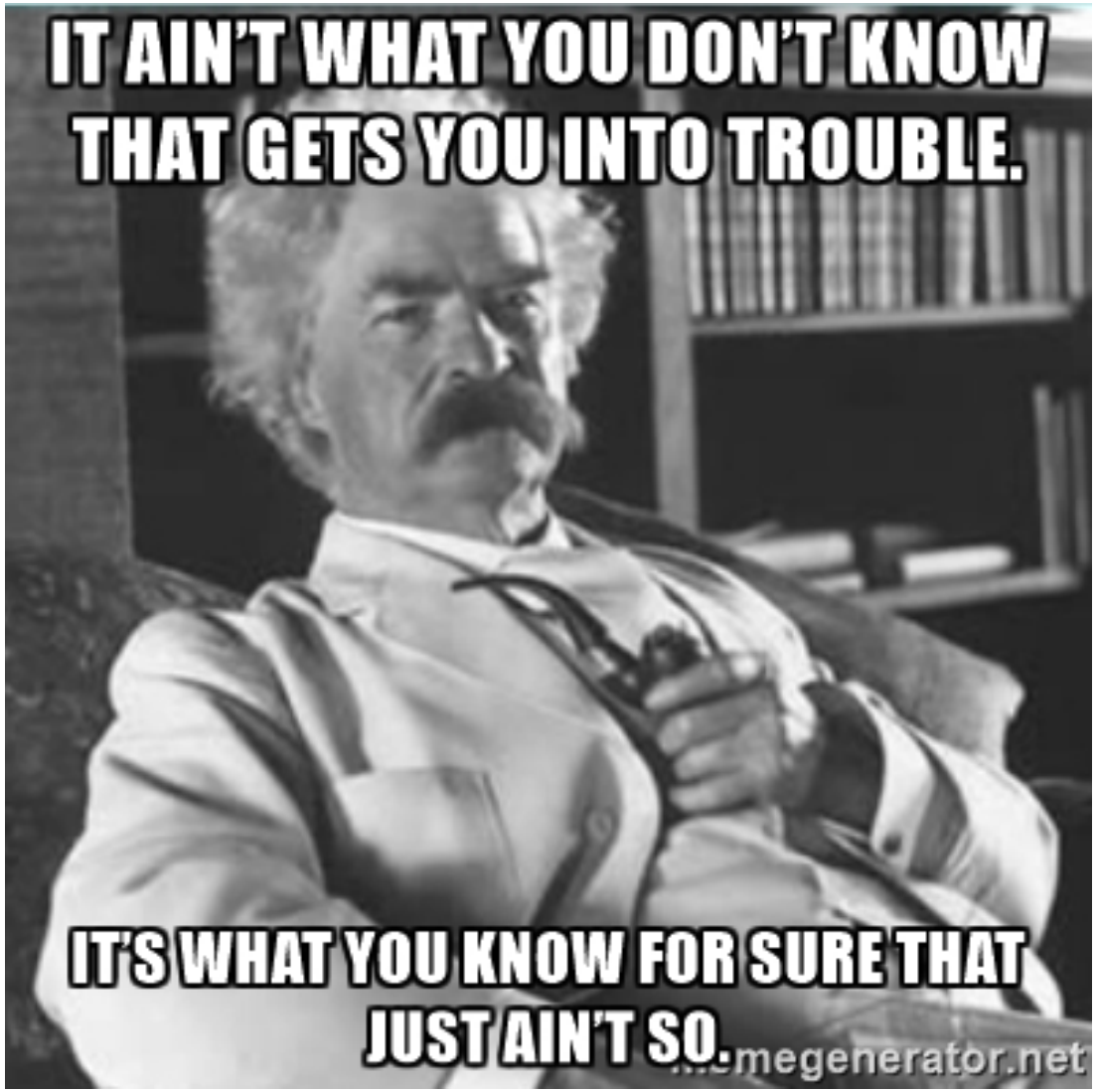
Thinking About Sequence

- **What can we ASSUME** about these statements in relation to each other?

```
int a, b;
...
a = 1;
b = 2;
```

NO logical dependencies, so can assume NOTHING about statements in relation to each other!

- ***ABSOLUTELY NOTHING***
  - *Cannot make any assumptions, because no logical dependencies exist between the two assignment statements*



Mark Twain (1835 – 1910)

NO assumptions can be made
between two statements that have **no logical dependencies**

Our "Best" Algorithmic Tools

- What Do We ***KNOW*** to be **True**?
- **Algorithmic Composition Tools**:
  1. **Logical Dependencies** (across statements, expressions)
  2. **Operator Precedence** (within expression)
  3. **Operator Associativity** (within expression)

SAFEST!
Most Elegant!
FASTEST!

What is our "Sledge-Hammer" when we get angry?

- *Mutexes, memory fences, critical sections, and similar*

(forced) **Synchronization**

- Is *expensive*
- Is *inelegant*
- Introduces surprising *corner-cases*
- Introduces surprising *constraints*
- May be necessary to **constrain your chosen algorithm** to enable correctness

Review: Logical Dependencies

- **Logical Dependencies** across statements-and-expressions **use the "resolved-version"** of a data object
  1. **Identify** the "version" of the data object
  2. **Resolve** value for that "version"
  3. **Use** "resolved-value"
  4. ***Profit!***

```
int a, b;
...
a = 1;
b = a + 3; // Which version of 'a'?
a = 5;
```

Don't use this version of a!

Use this version of a!

Don't use this version of a!

HINT: **Static Single Assignment** ("SSA-form") *internal to compiler* resolves each assignment to a new data object "version"

Review: Operator Associativity

- **Operator Associativity**: How operators of the same precedence are grouped (the order in which an operator's operands are evaluated)
  - Examples:
    - ***right-to-left:*** `operator=()`
    - ***left-to-right:*** `+, -, *, /`

Operators	Read from
Unary operators: ! ~ ++ -- + - * & (*typecast*) sizeof	Left to right
* / %	Right to left
+ –	Left to right
<< >	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= ^= |= <<= >=	Right to left
,	Left to right

FIRST **resolve this operand!**

SECOND **resolve this operand!**

```
int a, b=1;
int *p = &a;
*p = ++b;
// 'a' MUST be 2!  (always!)
...
```

Review: Operator Precedence

- **Operator Precedence**: The order in which operators are evaluated relative to each other
  - Example Precedence:
    - ***highest:*** (parenthesis)
    - ***higher:*** `*, /`
    - ***lower:*** `+, -`

	Operators	Read from
Higher	Unary operators: ! ~ ++ -- + - * & (*typecast*) sizeof	Left to right
	* / %	Right to left
	+ −	Left to right
	<< >	Left to right
	< <= > >=	Left to right
	== !=	Left to right
...	&	Left to right
	^	Left to right
	|	Left to right
	&&	Left to right
Lower	||	Left to right
	?:	Right to left
	= += -= *= /= %= &= ^= |= <<= >=	Right to left
	,	Left to right

MUST happen *FIRST*!

MUST happen *SECOND*!

```
int a, b;
...
a = 1;
b = 2 + a * 3;
// 'b' MUST be 5!  (always!)
...
```

Example 1: Physical Sequence

- **Source Code**: A **serialized form** of **physical sequences**

```
void MyTable::highlightRowCol(int mouse_x, int mouse_y)
{
  int row_index = computeRowIndex(mouse_y);
  int col_index = computeColIndex(mouse_x);

  bool is_row_locked = isRowLocked(row_index);
  bool is_col_locked = isColLocked(col_index);

  Color row_color = (is_row_locked) ? color_locked_ : color_unlocked_;
  Color col_color = (is_col_locked) ? color_locked_ : color_unlocked_;

  highlightRow_(row_index, row_color);
  highlightCol_(col_index, col_color);
}
```

- *Given:*

```
class Color { /*...*/ };

class MyTable
{
private:
  Color color_locked_;
  Color color_unlocked_;
  // ...
private:
  void highlightCol_(int col_index, Color color);
  void highlightRow_(int row_index, Color color);
public:
  // ...
  int computeColIndex(int mouse_x) const;
  int computeRowIndex(int mouse_x) const;

  void highlightRowCol(int mouse_x, int mouse_y);

  bool isColLocked(int col_index) const;
  bool isRowLocked(int row_index) const;
  // ...
};
```

- *(Erroneous)* **ASSUMPTION**: This is execution order.
  - ***Order appears*** to be ***explicit***
  - ***Order is tool*** we use to reason (so *some* guarantees *must* be present)
  - Assumption is reinforced by ***how programming is taught,*** and *(misunderstanding)* **programming language rules**.

NOT true!

BOLD assertion (defended on next slide)

Misunderstanding Programming Language Rules

- **ALL** programming languages have **rules**
  - **Syntax and semantics**
  - **Well-defined behavior**
- **ALL** rules establish **logical dependencies**
  - **Are how we reason** in that language!
  - Is **basis** for **Well-Defined Behavior** in that language!
- **ALL** programming languages are **SILENT on physical sequence**
  - Is **"Undefined Behavior" (UB) or "Unspecified Behavior"**
  - Is that which the **language CANNOT control** (e.g., CPU, memory access latencies)

Leads to idioms, patterns, conventions

Are the mechanisms that make the language work!

C++ Terms

Undefined Behavior:
Lack of Constraints
(order of globals initialization)

Unspecified Behavior:
Constraint Violation
(dereferencing NULL pointer)

- **NO programming language** mandates behavior for **physical sequence**, because that is **NOT POSSIBLE** (is not how compilers and CPUs work).
- If a programming language wants to **EXERT CONTROL**, it establishes **Logical Dependencies** (through which **Logical Sequences** can be expressed.)

ALL programming languages are designed to ACTUALLY execute on a CPU!

Revisit Example 1: Is Two Logical Sequences

(Previous)

In this case, logical sequences are totally unrelated!

```
void MyTable::highlightRowCol(int mouse_x, int mouse_y)
{
  int row_index = computeRowIndex(mouse_y);
  bool is_row_locked = isRowLocked(row_index);
  Color row_color = (is_row_locked) ? color_locked_ : color_unlocked_;
  highlightRow_(row_index, row_color);

  int col_index = computeColIndex(mouse_x);
  bool is_col_locked = isColLocked(col_index);
  Color col_color = (is_col_locked) ? color_locked_ : color_unlocked_;
  highlightCol_(col_index, col_color);
}
```

```
void MyTable::highlightRowCol(int mouse_x, int mouse_y)
{
  int row_index = computeRowIndex(mouse_y);
  int col_index = computeColIndex(mouse_x);

  bool is_row_locked = isRowLocked(row_index);
  bool is_col_locked = isColLocked(col_index);

  Color row_color = (is_row_locked) ? color_locked_ : color_unlocked_;
  Color col_color = (is_col_locked) ? color_locked_ : color_unlocked_;

  highlightRow_(row_index, row_color);
  highlightCol_(col_index, col_color);
}
```

- **By tracing Logical Dependencies** (by tracing C++ Well-Defined Behavior)
  - **We Discover** (in this case): Two logical sequences are present!
  - These **separate logical sequences** have NOTHING to do with each other!
    - In this case, **could** even **be separate functions**!

Example 2: Physical Sequence

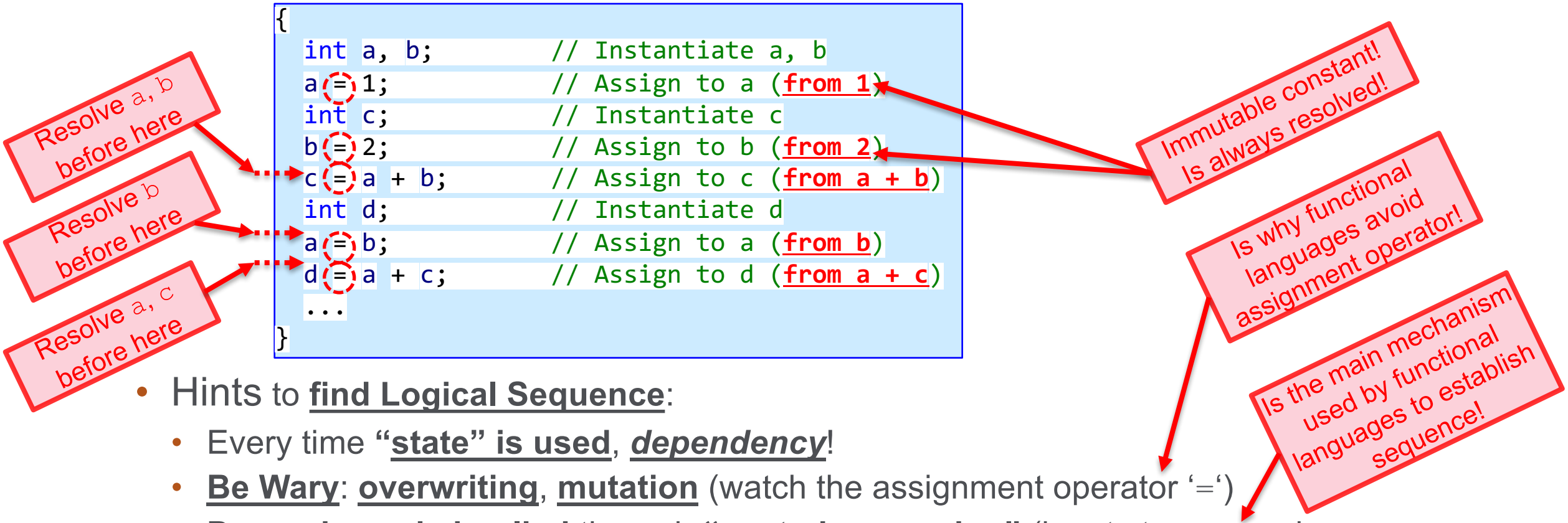
- **How** to **identify logical sequence(s)**?

```
{
  int a, b;          // Instantiate a, b
  a = 1;             // Assign to a (from 1)
  int c;             // Instantiate c
  b = 2;             // Assign to b (from 2)
  c = a + b;         // Assign to c (from a + b)
  int d;             // Instantiate d
  a = b;             // Assign to a (from b)
  d = a + c;         // Assign to d (from a + c)
  ...
}
```

- A **more abstract** (perhaps "real-world") **example**
  - Code seems …*indirect*
  - Variable names are …*vague*
  - Comments are …*unhelpful*

Example 2: Logical Sequence (inferring)

- **Logical Sequence** can **be inferred** from physical sequence (using programming language "Well-Defined" behavior)



- Hints to **find Logical Sequence**:
  - Every time **"state" is used**, ***dependency***!
  - **Be Wary**: **overwriting**, **mutation** (watch the assignment operator '=')
  - **Dependency is implied** through **"nested-expansion"** (inputs to expressions, function-calls)

State: Hidden Dependencies!

- "**State**" is "**tricky**"
  - **WHEN** did you use it? Was it **CORRECT**?
    - Was it "**stale**"? *(too "old")*
    - Was it "**corrupted**"? *(update was in-progress)*
    - Was it "**too-new**"? *(race condition)*
- ***Two*** issues exist:
  1. **Compute State** *(e.g., expressions with operands)*
     - Were **operand values** "correct"?
  2. **Change State** *(e.g., overwrite/mutate previous state)*
     - **WHEN** did that occur? *(stale, corrupted, too-new?)*

`const` (read-only) makes this **simple**! (Value is always correct!)

Not too hard

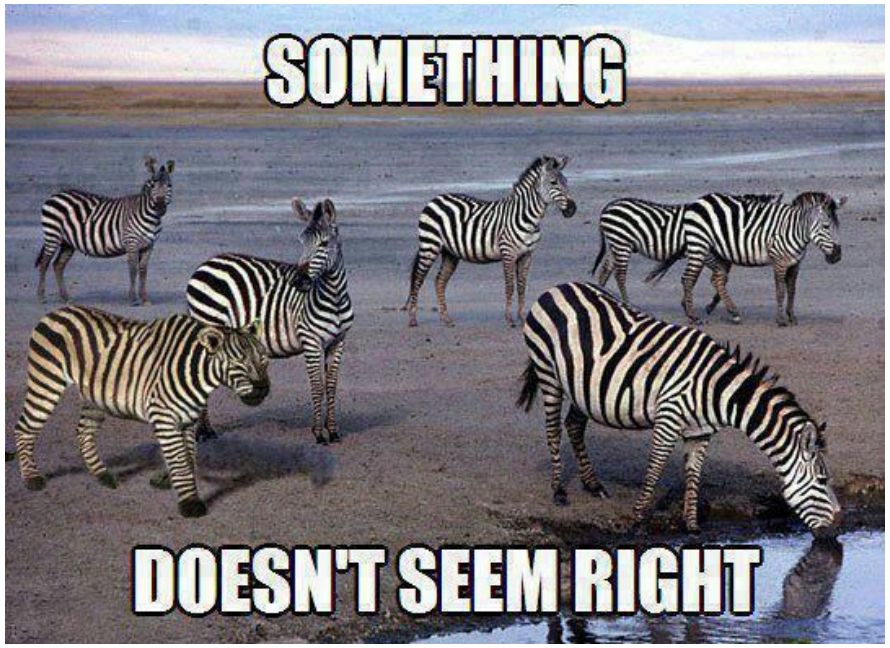
Much Trickier!

Anytime you **use state**, there **might be** hidden **dependencies**! So,
"Use state with care!"

Assignment '=' Is A Change In State

- Assignment `operator=()`: Tricky!
  1. **Dependencies** to compute **RHS**
  2. **Overwrite LHS** (Dependencies for WHEN did that occur relative to other reads/writes?)
- **Functional** programming **avoids** `operator=()`
  - **Avoids (2)**, but **still has (1)**
  - **Addresses (1)** by **functionally computing** all values from "First Principles"

    (i.e., "State Is Evil")



Functional Programming Mantra:

"State Is Evil,

all values are computed from First Principles"

“Resolving” Dependencies

- **State** (values)
  1. **Constant**: `42`
     - Easy! Done at compile!
  2. **Computed**: `a * b`
     ① Must complete evaluation before is used (i.e., “RHS”)
  3. **Overwritten**: `c = a * b`
     ① Must complete evaluation of RHS
     ② Must overwrite LHS
  4. **Mutated**: `c += a * b`
     ① Must resolve LHS
     ② Must resolve “delta” (e.g., “RHS”)
     ③ Must overwrite LHS

- **Side-Effects** *(tricky!)*
  1. Resolve “**before**” side-effect “**starts**”
  2. Resolve “**before**” side-effect “**completes**”
  3. Resolve “**after**” side-effect “**completes**”

Functional Programming (tries to) avoid these!

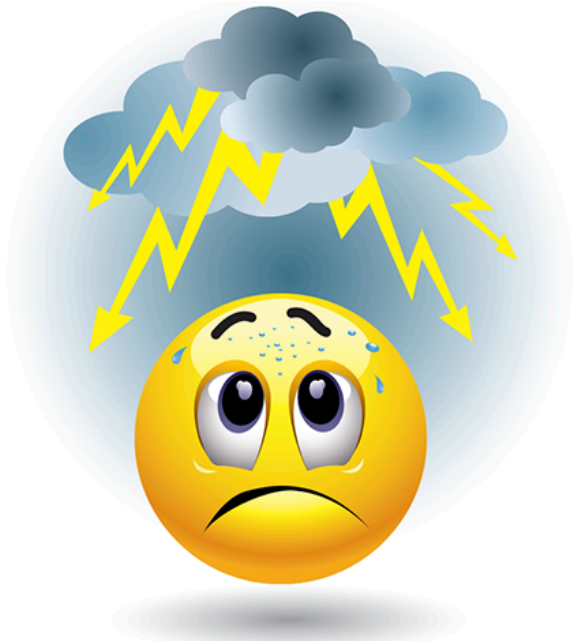
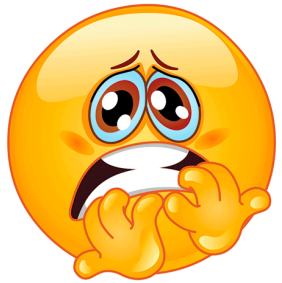
Reasoning about dependencies is actually **pretty simple**

In practice,
Complexity increases (non-linearly) with a greater number of dependencies

State: "Nervous Meter"





Care Free!

No data object	`const` data object	Computed data object	Overwritten data object	Mutated data object

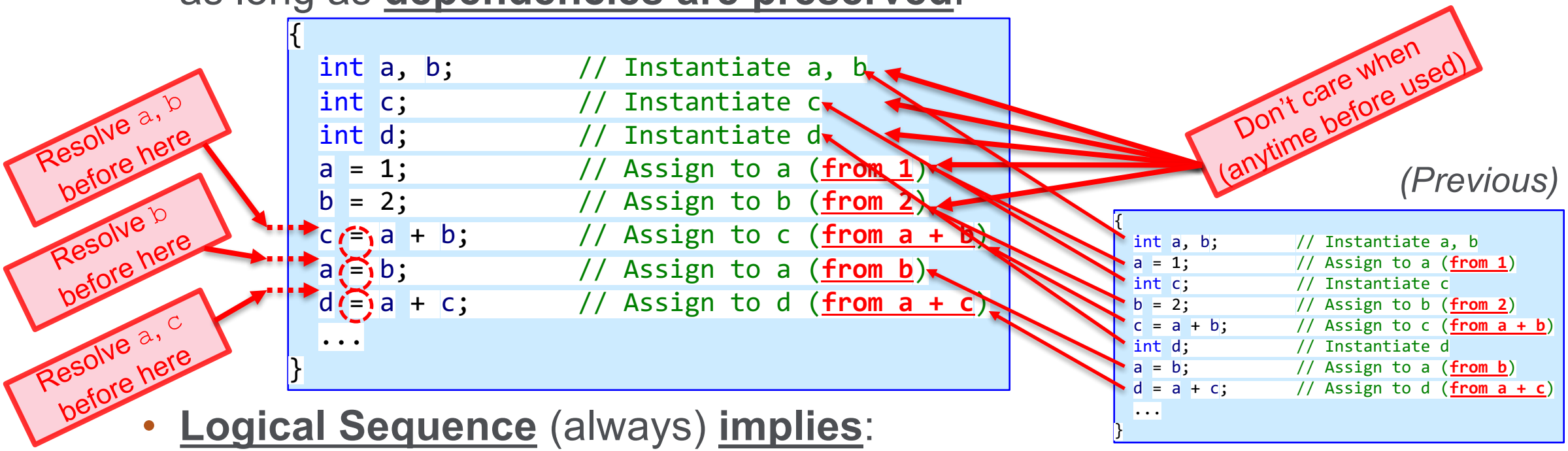
Very Nervous!

Functional Programming

All Programming

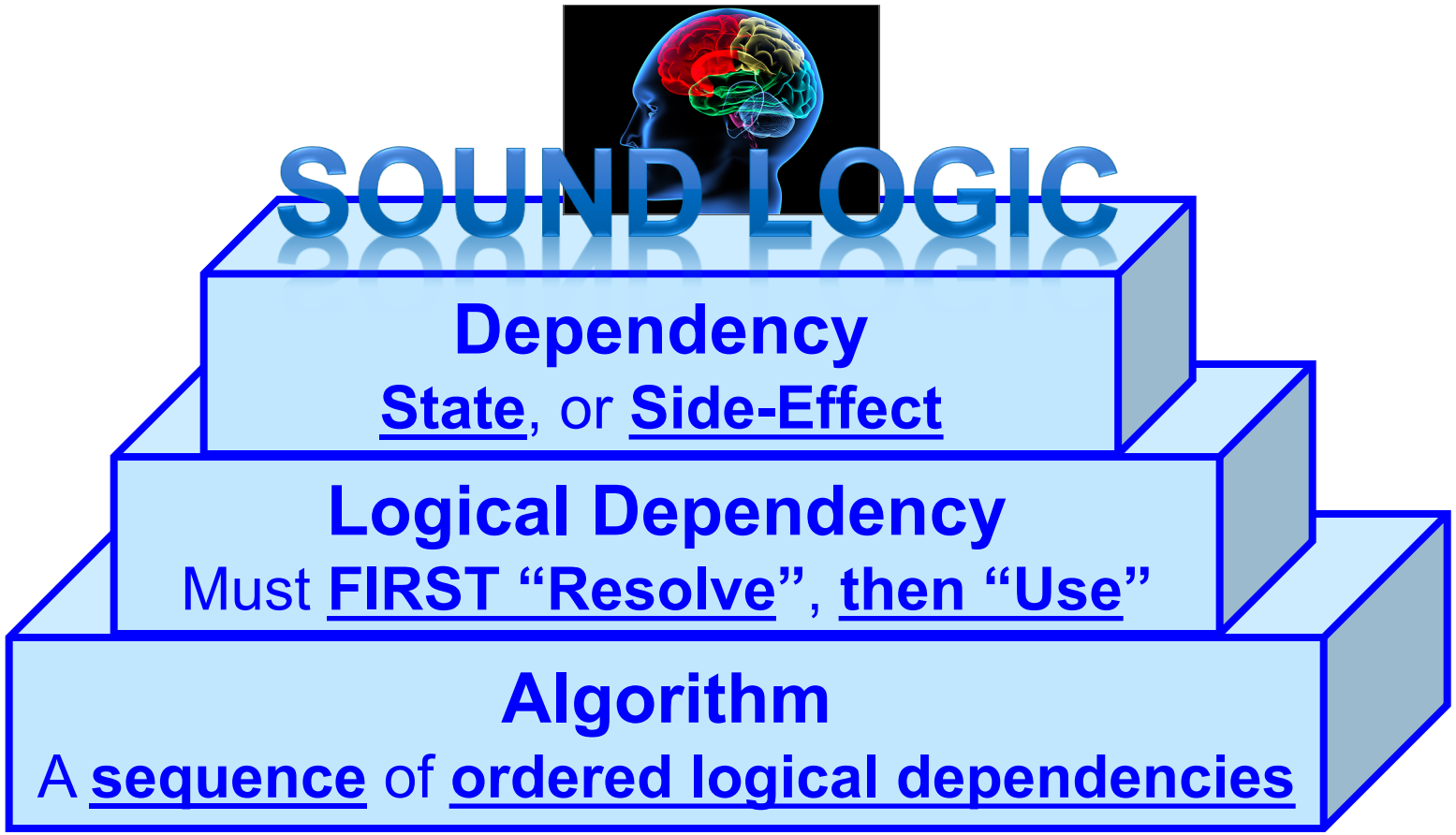
Example 2: Logical Sequence (preserving)

- **Logical sequence survives** *(arbitrary!)* physical sequence "**shuffling**", as long as **dependencies are preserved**.



- **Logical Sequence** (always) **implies**:
  - **State** must be "**resolved" before is used**
  - "**Zones" of *"Don't Care!"*** exist – we do not care about **physical** sequence order

Concept Review: Logical Sequence



Rule: **Any order** that **respects logical dependencies** is **logically equivalent**

Concept Review: Physical Sequence

- **Physical** Sequence: **Don't care**.
  - Is **arbitrary** and weird
  - **Moves around** for seemingly no reason whatsoever
    - At **compile-time**, at **runtime**
- When you hear the word "Sequence", IMMEDIATELY ask, "Logical Sequence"?
  - Because if it's **not a logical sequence**, **we don't care**.

Parks And Recreation (2009-2015)

Physical Sequence:
No dependencies exist, so
order CANNOT be enforced.

The C++ "As If" Rule

Conspiring to change your code *(unbeknownst to you)*

The C++ “As If” Rule (paraphrased)

- “The Chillin’ Rule”:

“It’s All Good.”

The C++ “As If” Rule (paraphrased)

- “The Cool Rule”:

“Relax. I got this.”

The C++ “As If” Rule (paraphrased)

- “The Casual Rule”:

“Dependencies respected. Everything else is casual.”

The C++ “As If” Rule (quoted)

C++11 Standard, §1.9/1

The semantic descriptions in this International Standard **define a** parameterized nondeterministic **abstract machine**. This International Standard places **no requirement on the structure** of conforming implementations.

<snip>, (emphasis added)

- The **C++ Standard** is **TWO things**:

1. **Explanation of (observable) behavior** *(which IS the standard!)*
   - Must be respected! Is ***Well-Defined Behavior!***
2. **Definition of an abstract machine** to *explain* that behavior *(including sufficient structure to demonstrate the machine can be constructed)*
   - Implementation is free to disregard! Is ***Artifact!***

The C++ “As-If” Rule (In Practice)

- The C++ compiler and CPU are ***permitted*** to perform **any and all code transformations** that do not change the program’s observable behavior.

Translation:
The **Compiler** and **CPU** can **do whatever they want**, as long as **logical dependencies are respected**.

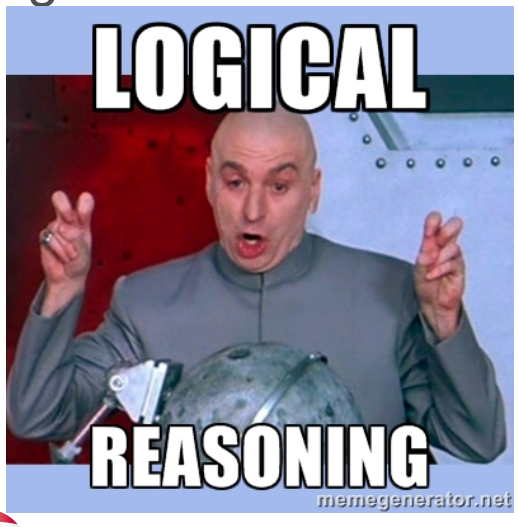
Corollary:
We should ***expect*** the compiler and CPU to reorder the execution of our C++ statements at some level

Compare "C++ As-If" With Other Languages

- **C Language** has the equivalence of the **"As-If" rule**
  - Lacks some subtle rules to gain intent (e.g., "copy elision")

Also designed to run on an actual CPU!

- Many **other languages** ("simpler"): Language specification merely **documents the intended behavior** of the implementations (including that which is explicitly unspecified) in the target intermediate representation specification
  - Might have a **"reference implementation"** that is "correct", where differing behavior of all other implementations are "wrong"
- **Some languages** strictly **specify formal language rules** without mentioning intent
  - Is similar to "As-If" rule when behavior is not explicitly specified



Intent != Behavior

C++ Expresses “Intent”

- std-discussion@isocpp.org
  - Richard Hodges, “Re: [std-discussion] throw std::exception with stack trace (portable)”
  - Sat-16-Apr-2016, https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/A17G1ram9ns

<snip>,

“C++ is not like other languages. It expresses intent. The compiler transforms that intent into 'as if' code.

It is wiser to focus efforts on guaranteeing that the correct intent is specified.”

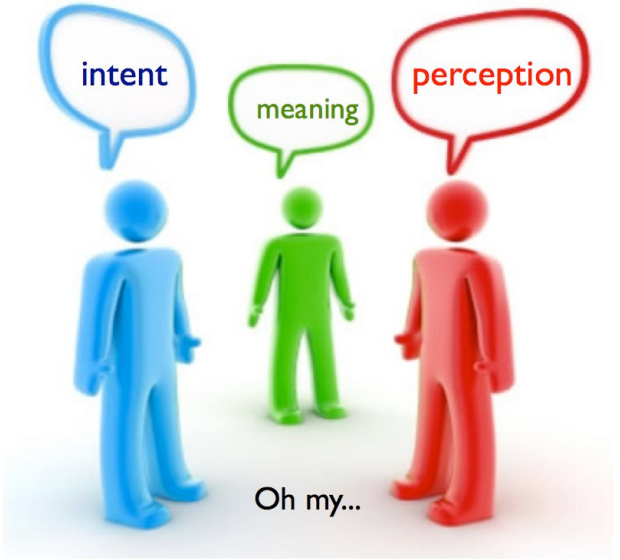
<snip>

C++ is designed to run *(efficiently!)* on an **actual CPU**!

Any “extra-sequence-guarantees” offered by other languages impose *(significant!)* efficiency penalties on the compiler, and CPU!

“Intent” is expressed through C++ Well-Defined behavior to **establish dependencies** within the algorithm.

Expressing "Intent"



- All **languages express "intent"** (somehow)
  - Language rules for "Well-Defined" Behavior
- **Intent allows "Optimizations"**
  - Respect "intent"; **Allow ANY AND ALL changes** that do **not undermine "intent"**
    - Copy Elision
    - Return-value optimization (RVO)
    - Function Inlining
- All languages **run on** an **actual CPU**
  - Possibly with many levels of "indirection", "**translation**", "interpretation"
  - Dependencies ("intent") are respected

Controlling for "intent" permits **MAXIMUM FREEDOM** to perform **optimizations**! *(Does not control for "artifacts")*

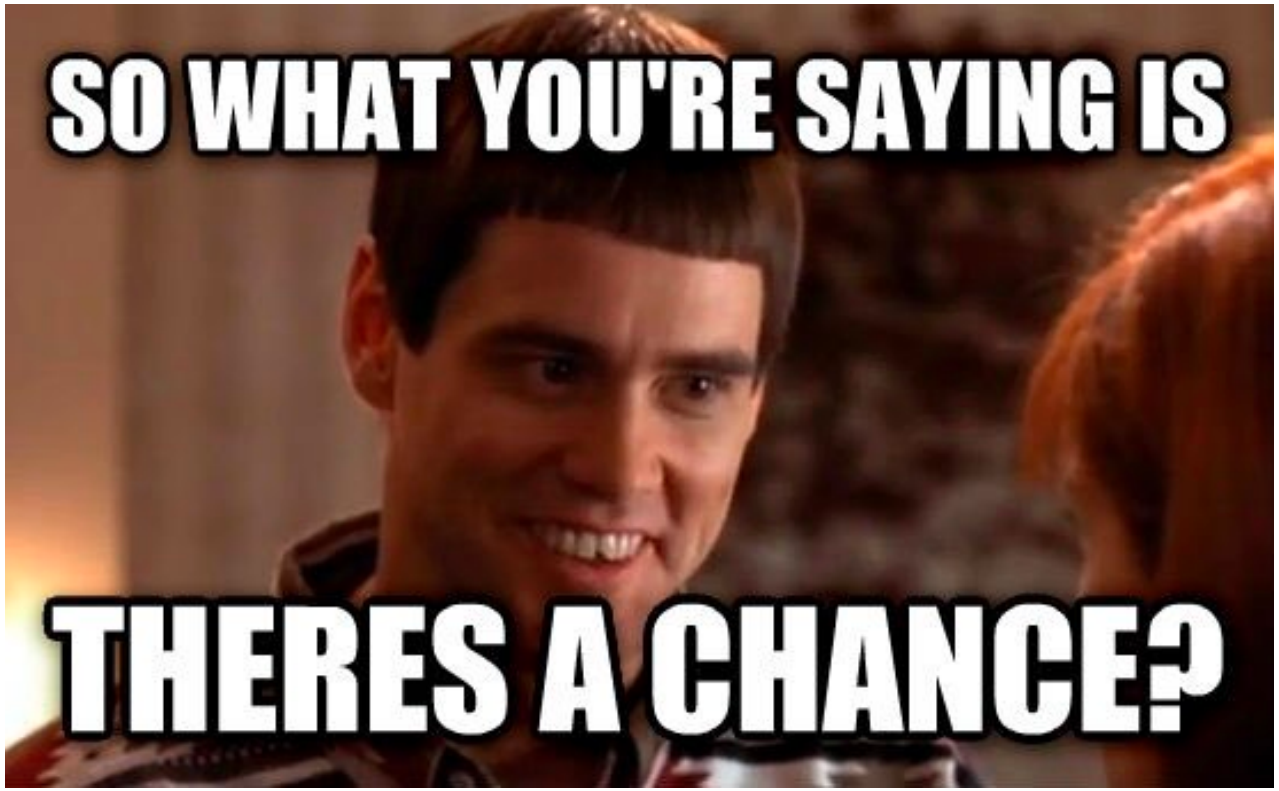
Algorithmic "Intent" is expressed by establishing (logical) dependencies

Summary
The “take-away for today”

Stop Thinking Imperatively (don’t assume order)

Instruction Order Summary

- There is “**a chance**” that your statements **execute in the order you specify** in your C++ source code file.

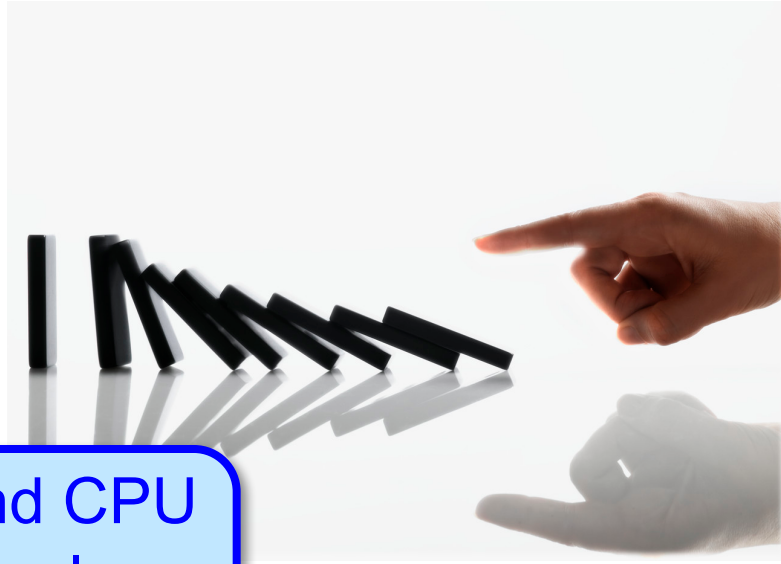


Dumb and Dumber (1994)

- …But ***not a very good one***.

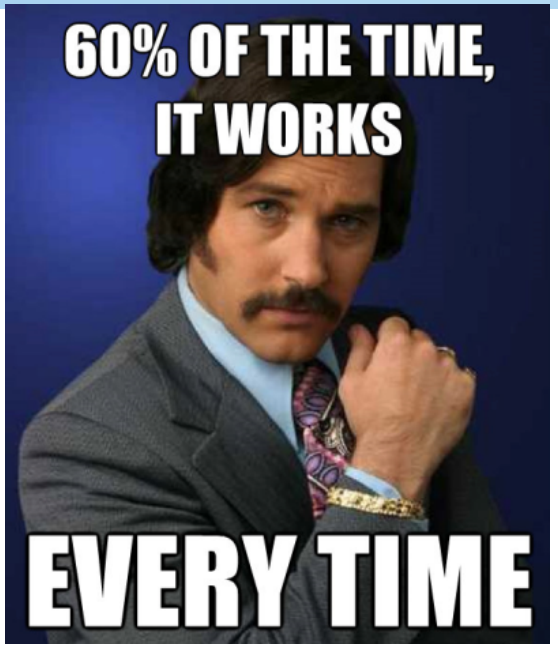
Logical Sequence: Defined by dependency

- **Any order** that respects dependency order is logically equivalent.
  - **Logical Sequence** is **defined by dependencies** *(we care!)*
  - We care **not at all** about any other order
- With explicit **focus on dependency management** in design and implementation:
  - Our single-threaded implementations are *faster*
  - Our multi-threaded designs are *easier*
  - Many irrelevant details disappear from discussion
  - We best exploit technologies within compiler and CPU



Granting **maximum latitude** to the compiler and CPU regarding order **leverages** the underlying technology to get the **fastest and most efficient** programs

We Control For Dependencies



- **Dependencies are the basis** for how we reason
  - **Dependencies are respected** by the compiler and CPU (or all is lost)
  - If **no dependencies** present, **order is uncontrolled**
- The **C++ Language**, the **compiler**, and the **CPU** are based on well-defined behavior to **manage dependencies** *(not "instructions", not "statements")*
  - They all **AGREE** on what are **"dependencies"**
  - They all **DISAGREE** on what is an **"instruction" or "statement"** *(no commonality)*
- Defining (or relying upon) **"physical order"** is **problematic** – don't try

Why would ANYONE think "instructions" (or statements") would execute "in-order"? It is **HIGHLY DESIRABLE** that the order be ***Uncontrolled***!

Dependencies: How We Design, Implement

1. Make **No Assumptions regarding "order"**
   (dependencies are respected, "order" is not)

2. **Over-Specifications restrict efficiencies**
   (specify only those minimal dependencies required)

3. *C++ As-If Rule:*
   The **Compiler** and **CPU** can **do whatever they want**,
   as long as **logical dependencies are respected**

A Bonus: By focusing on dependencies, our design often changes
- More **scalable**, **flexible**, **efficient**, **simpler** *(fewer assumptions)*

Essence Of Parallel, Concurrent

Best Practice:
Go **Parallel when you can**,
Concurrent when you must

1 *Essence of **Parallel**:*
No Dependencies
(is about leveraging the hardware)

Design to be simple and obvious!

2 *Essence of **Concurrent**:*
Interacting Dependencies
(is about managing complexity)

Design to algorithm desired!

Well-Defined Behavior

Well-Defined Behavior
- Comes from ***logical*** sequence
  - Defined by logical dependencies

Respected! Enforced!

Undefined Behavior is **always repaired** by establishing **logical dependencies**!

Undefined Behavior
- Comes from ***physical*** sequence
  - No dependencies exist, so order *cannot* be enforced

Rodney Dangerfield

November 22, 1921 – October 5, 2004

Patron Saint of Physical Sequence

The Lego Movie (2014)

Questions?