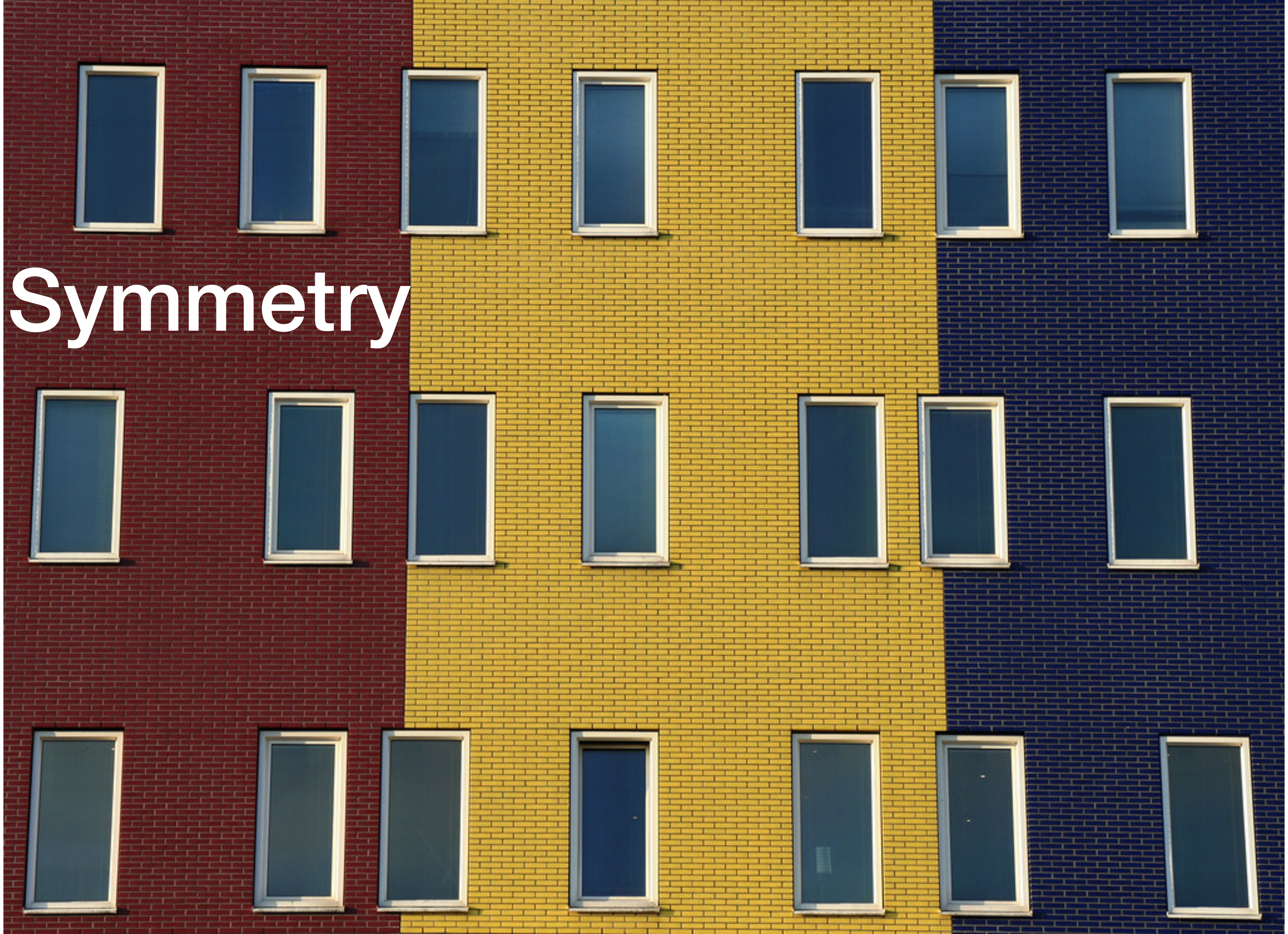
## Abstract

Why should we be concerned with symmetry? Symmetry is fascinating to the human mind and everyone likes objects or patterns that are in some way symmetrical. It is an interesting fact that nature often exhibits certain kinds of symmetry in the objects and phenomena in our Universe.

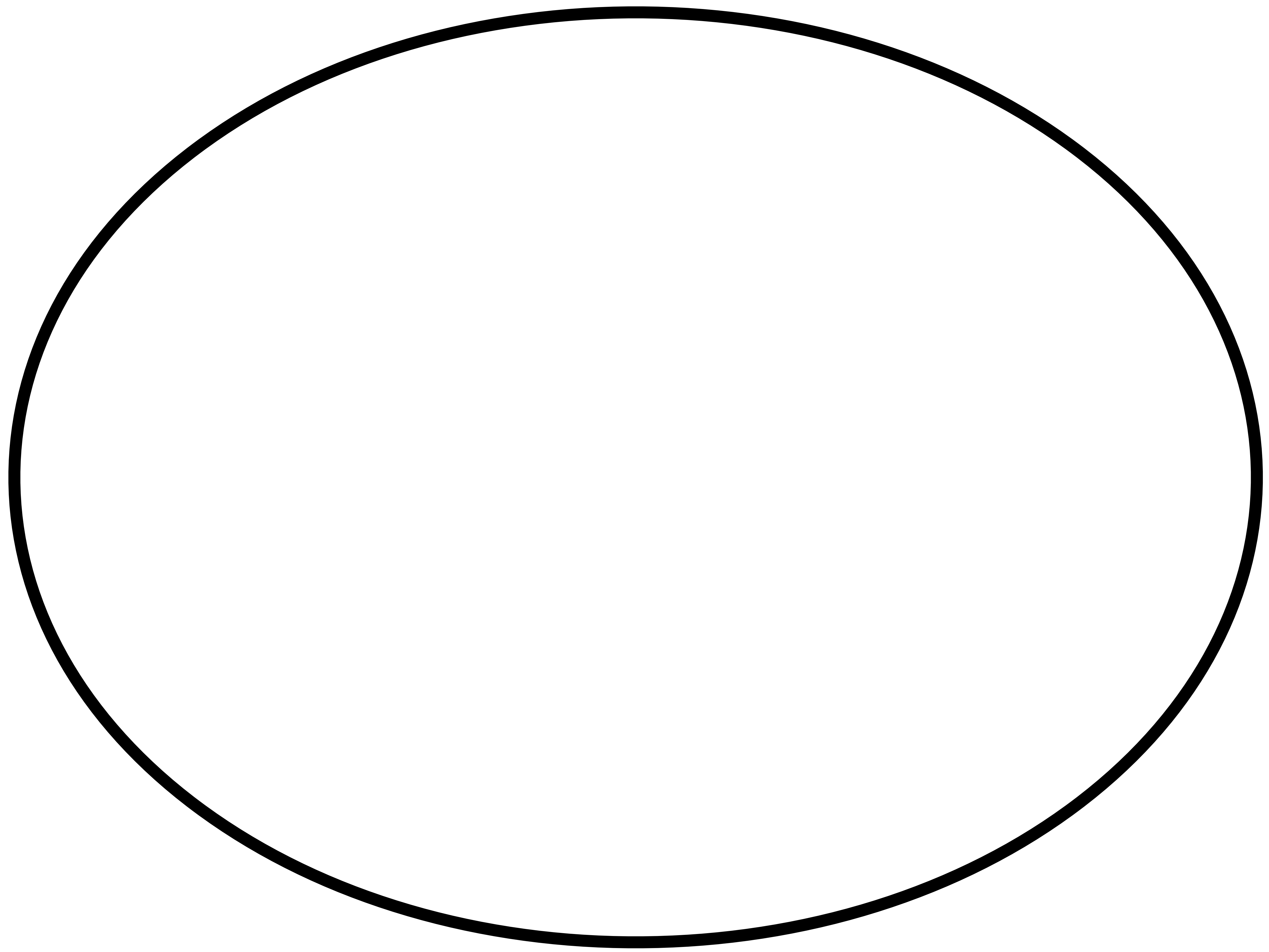
We have, in our minds, a tendency to accept symmetry as some kind of perfection. Yet it so often eludes us…

Let’s look at code and see what interesting properties emerge from various kinds of symmetries. A quest for the ‘Character of Code’, following Richard Feynman’s awe-inspiring take on physical laws.

We'll be looking to identify patterns in code, interested to see when such patterns exhibit some sort of symmetry that is advantageous in some way for reliability, performance, maintenance and discoverability.

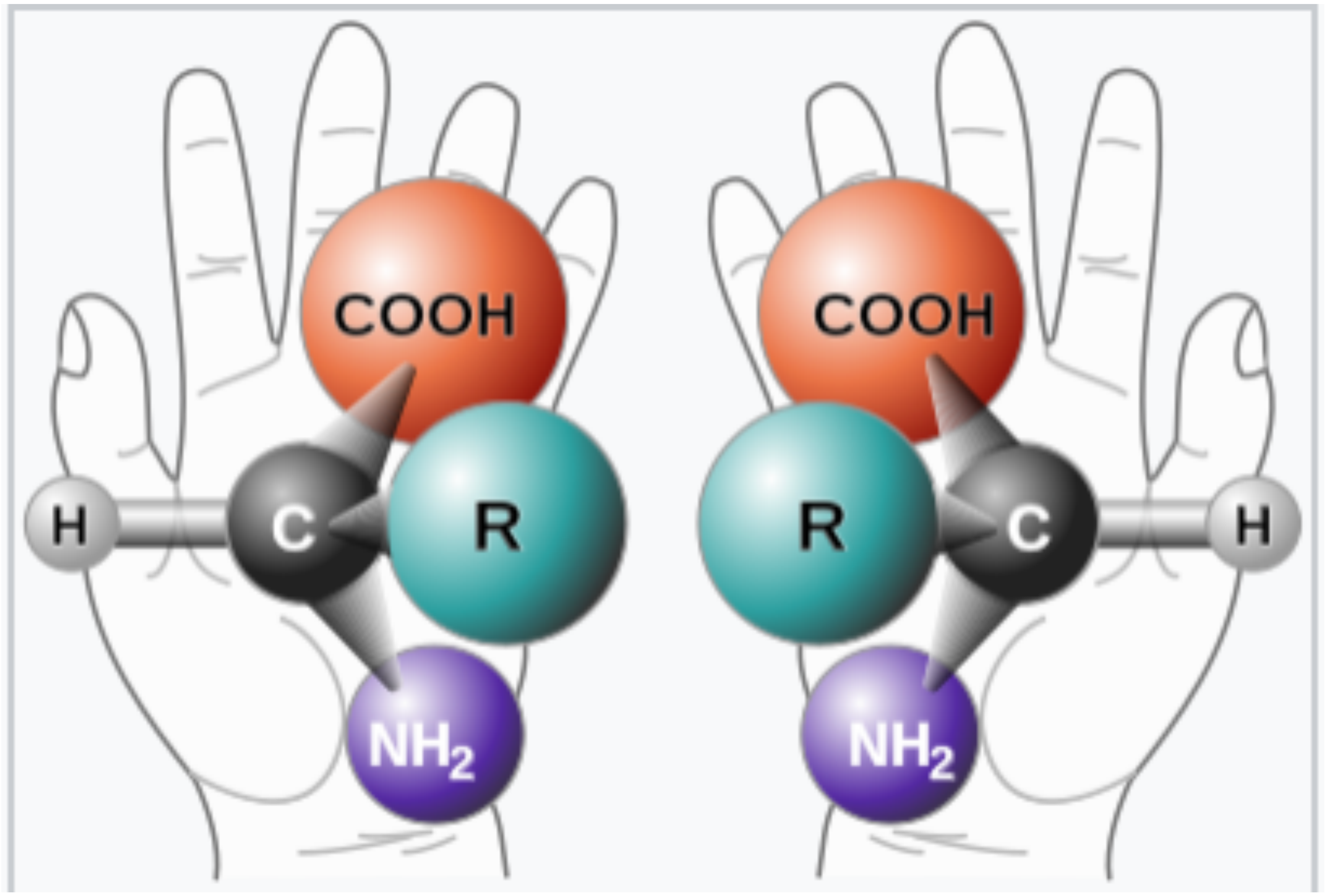


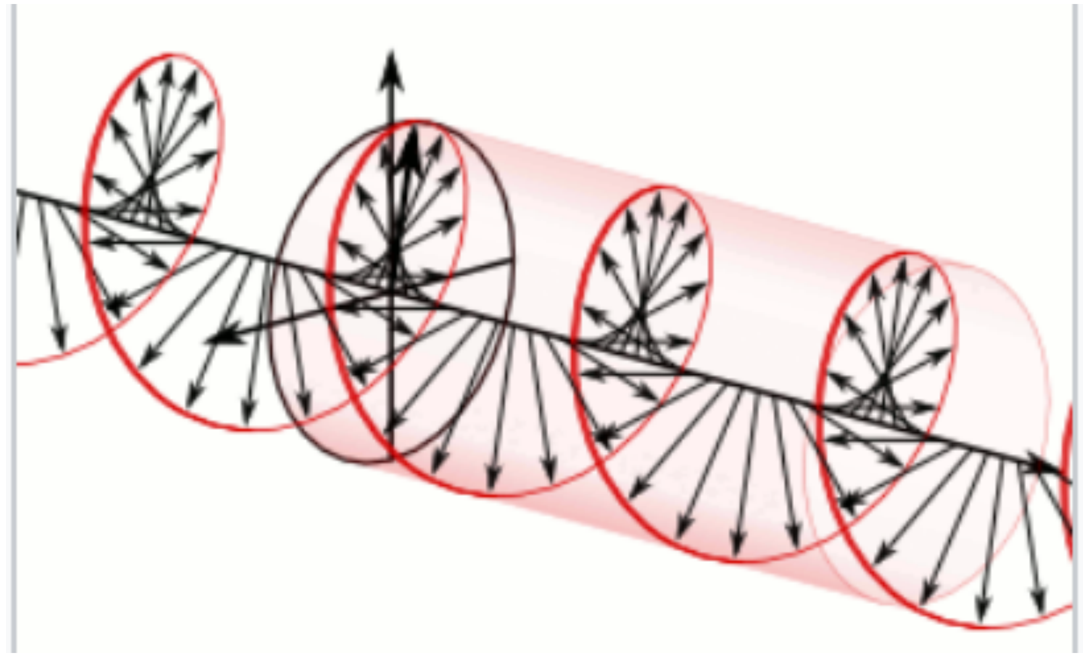
## Symmetry



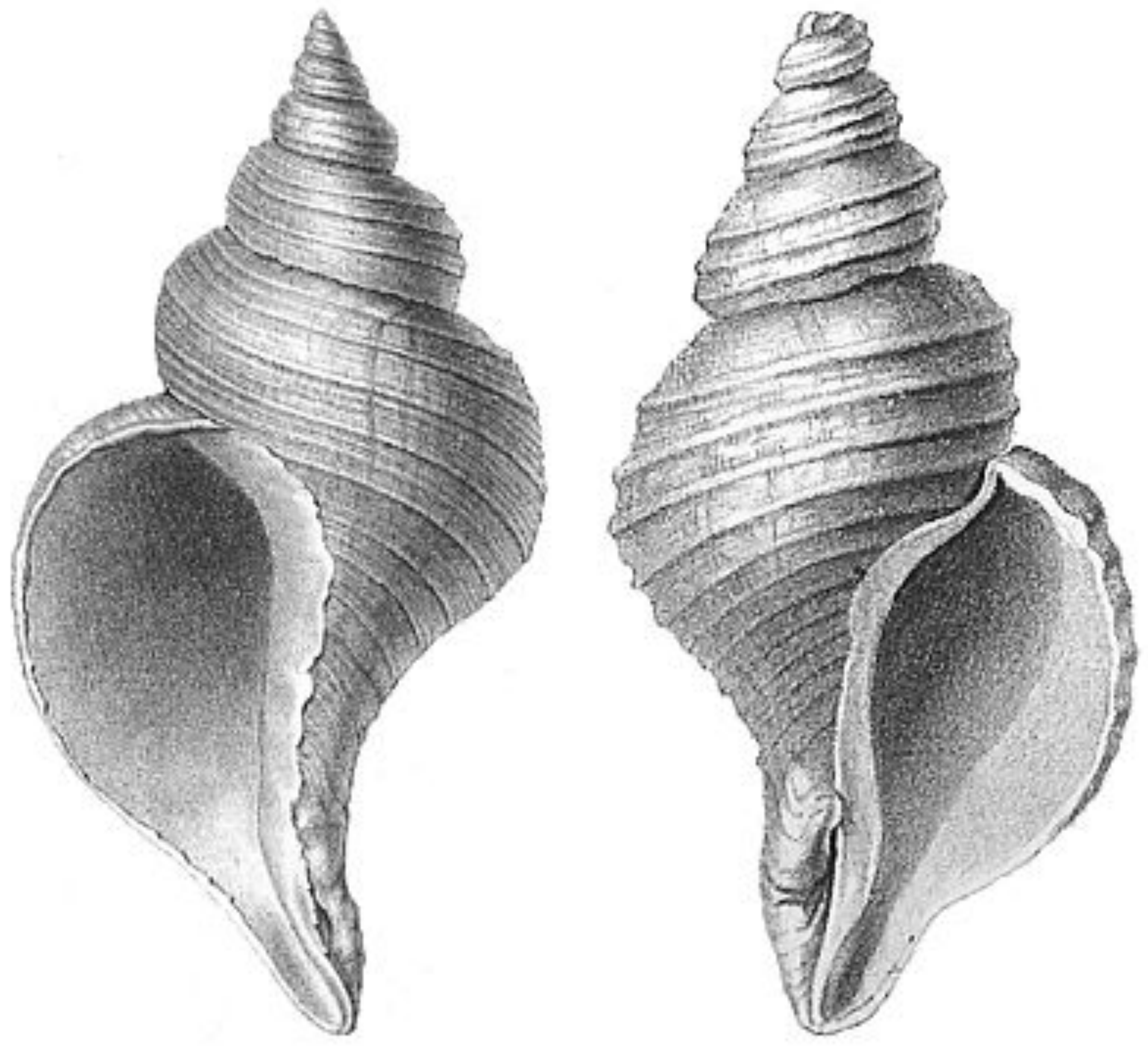
In fact, it is like the old idea of ancient GREEKS that circles were perfect, and it was rather horrible for them to believe that the planetary orbits were not circles, but only nearly circles.

# Chirality



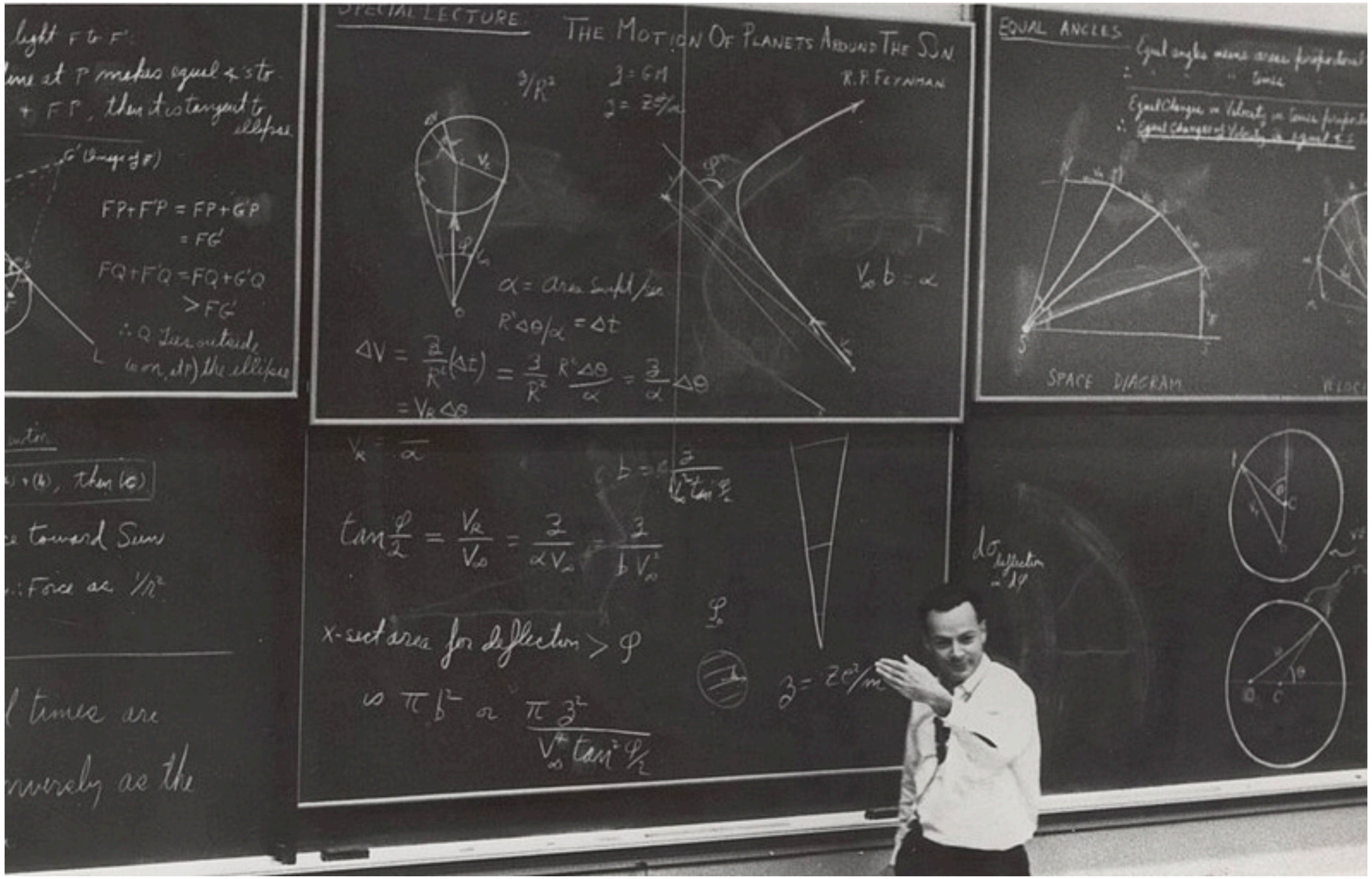


rotation of plane polarized light by chiral substances



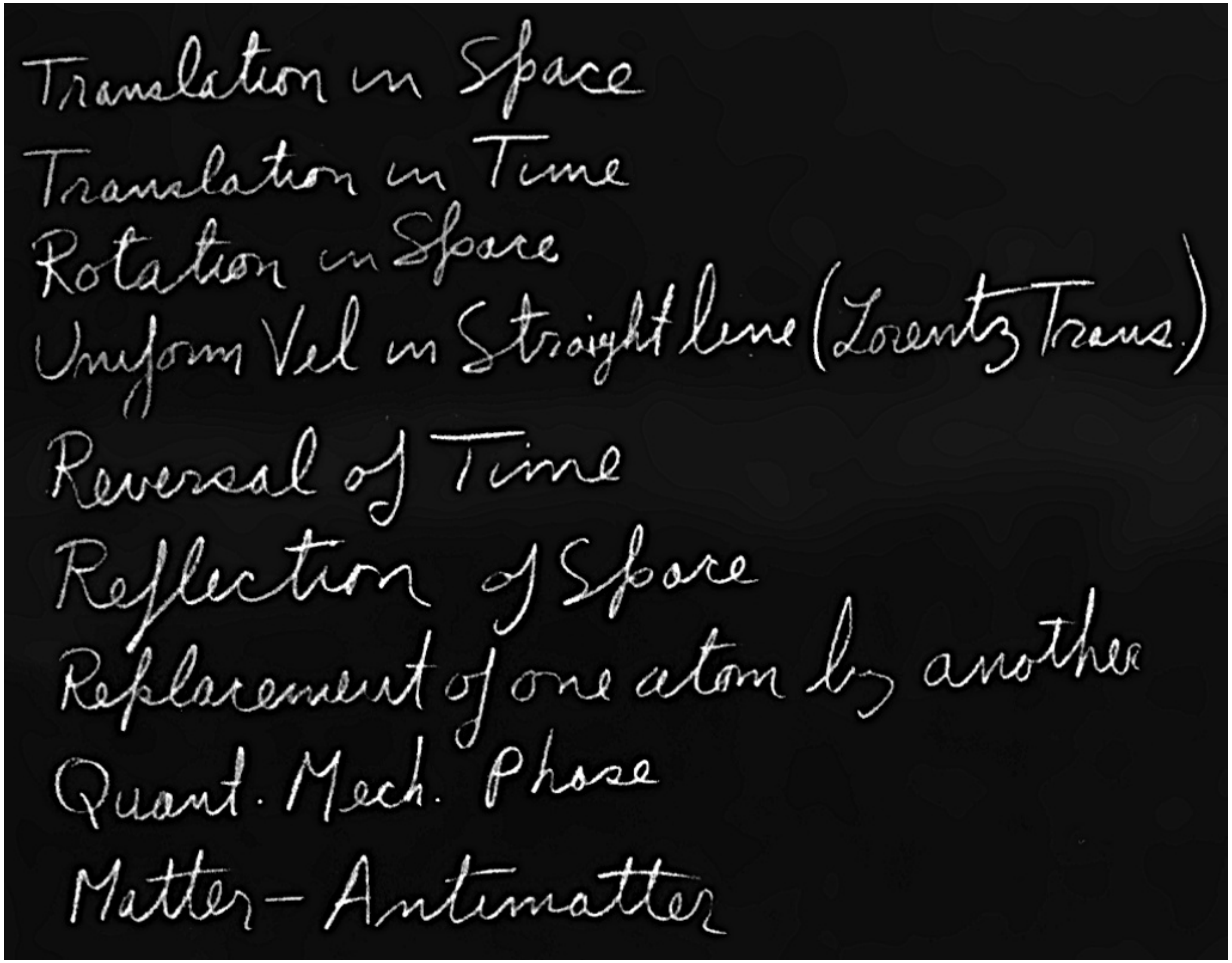
(left-handed) Neptunea angulata | (right-handed) Neptunea despecta

## Richard Feynman - The Character of Physical Law (1964)


SPECIAL LECTURE
THE MOTION OF PLANETS AROUND THE SUN
R.P. FEYNMAN
EQUAL ANGLES
SPACE DIAGRAM
FP+F'P = FP+G'P
= FG'
FQ+F'Q = FQ+G'Q
> FG'
$\alpha$ = area swept/sec
$\tan\frac{\varphi}{2} = \frac{V_R}{V_\infty}$
x-sect area for deflection > $\varphi$
toward Sun
∴ Force as $1/R^2$

youtube.com/watch?v=kEx-gRfuhhk

# Symmetry in Physical Laws



Translation in Space
Translation in Time
Rotation in Space
Uniform Vel in Straight line (Lorentz Trans.)
Reversal of Time
Reflection of Space
Replacement of one atom by another
Quant. Mech. Phase
Matter - Antimatter

Richard Feynman - The Character of Physical Law
Part 4: Symmetry in Physical Laws

youtube.com/watch?v=tGsYbK-Chkg

## Symmetry beyond geometry

Symmetry goes way beyond simple geometrical shapes & patterns.

Symmetry is not just about **observing** the properties of objects, but also for *transformations*:

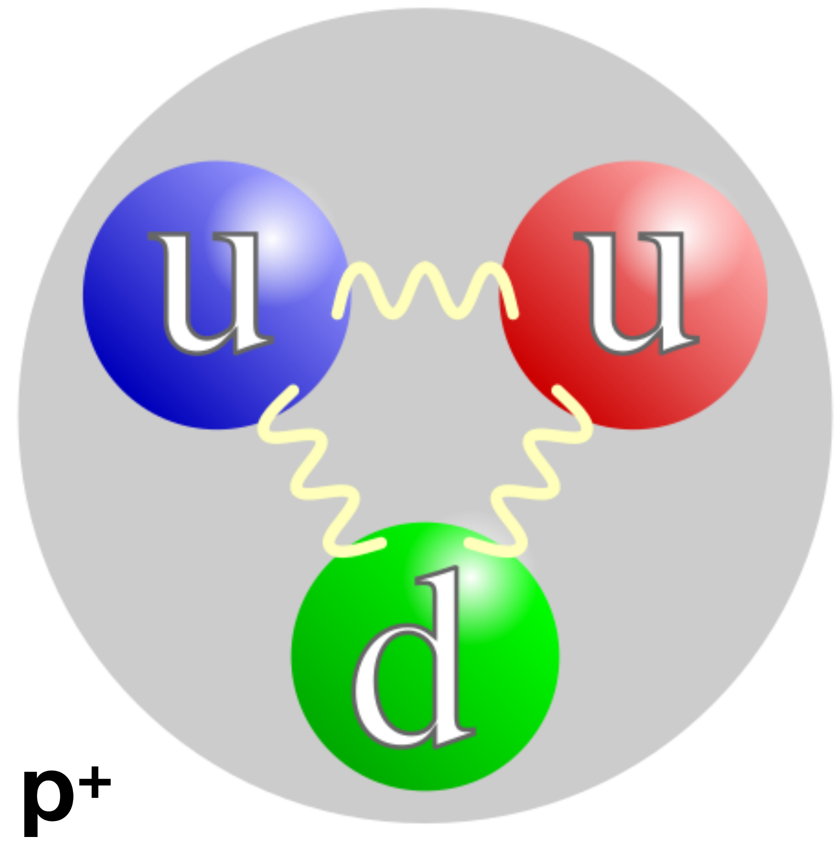
- what can you **do** to a symmetrical object so it can "looks" the same

He’s the first mathematician to study symmetry for non-geometric entities (eg. equations, functions, polynomials, groups).

ted.com/talks/marcus_du_sautoy_symmetry_reality_s_riddle

Beauty, truth and ... physics?

1,527,719 views | Murray Gell-Mann | TED2007 • March 2007



"Are elegant equations more likely to be right than inelegant ones?"

“Beauty is a very successful criterion for choosing the right theory”

ted.com/talks/murray_gell_mann

$$\frac{\partial E_x}{\partial x} + \frac{\partial E_y}{\partial y} + \frac{\partial E_z}{\partial z} = 4\pi\rho \qquad (1)$$

$$\frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} = 0 \qquad (2)$$

$$\left.\begin{aligned} \frac{\partial E_x}{\partial x} - \frac{\partial E_y}{\partial y} + \frac{1}{c}\dot{B}_z &= 0 \\ \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} + \frac{1}{c}\dot{B}_x &= 0 \\ \frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} + \frac{1}{c}\dot{B}_y &= 0 \end{aligned}\right\} \qquad (3)$$

$$\left.\begin{aligned} \frac{\partial B_x}{\partial y} - \frac{\partial B_y}{\partial x} - \frac{1}{c}\dot{E}_z &= \frac{4\pi}{c} j_z \\ \frac{\partial B_y}{\partial z} - \frac{\partial B_z}{\partial y} - \frac{1}{c}\dot{E}_x &= \frac{4\pi}{c} j_x \\ \frac{\partial B_z}{\partial x} - \frac{\partial B_x}{\partial z} - \frac{1}{c}\dot{E}_y &= \frac{4\pi}{c} j_y \end{aligned}\right\} \qquad (4)$$

Original form

---

$$\nabla \cdot \mathbf{E} = 4\pi\rho \qquad (1)$$

$$\nabla \cdot \mathbf{B} = 0 \qquad (2)$$

$$\nabla \times \mathbf{E} + \frac{1}{c}\dot{\mathbf{B}} = 0 \qquad (3)$$

$$\nabla \times \mathbf{B} - \frac{1}{c}\dot{\mathbf{E}} = \frac{4\pi}{c}\mathbf{j} \qquad (4)$$

Simplified using rotational symmetry

---

$$\partial_\nu F^{\mu\nu} = \frac{4\pi}{c} j^\mu \qquad \text{(1 and 4)}$$

$$\varepsilon^{\mu\nu\kappa\lambda}\partial_\nu F_{\kappa\lambda} = 0 \qquad \text{(2 and 3)}$$

Further simplified using the symmetry of special relativity

# Symmetry in Code
## ¿Should We Care?

**Meeting C++**

November 2024

@ciura_victor
@ciura_victor@hachyderm.io
@ciuravictor.bsky.social

**Victor Ciura**
Principal Engineer
Rust Tooling @ Microsoft

## About me

Advanced Installer

Clang Power Tools

Oxidizer SDK

Visual C++

Rust Tooling

🐦 @ciura_victor

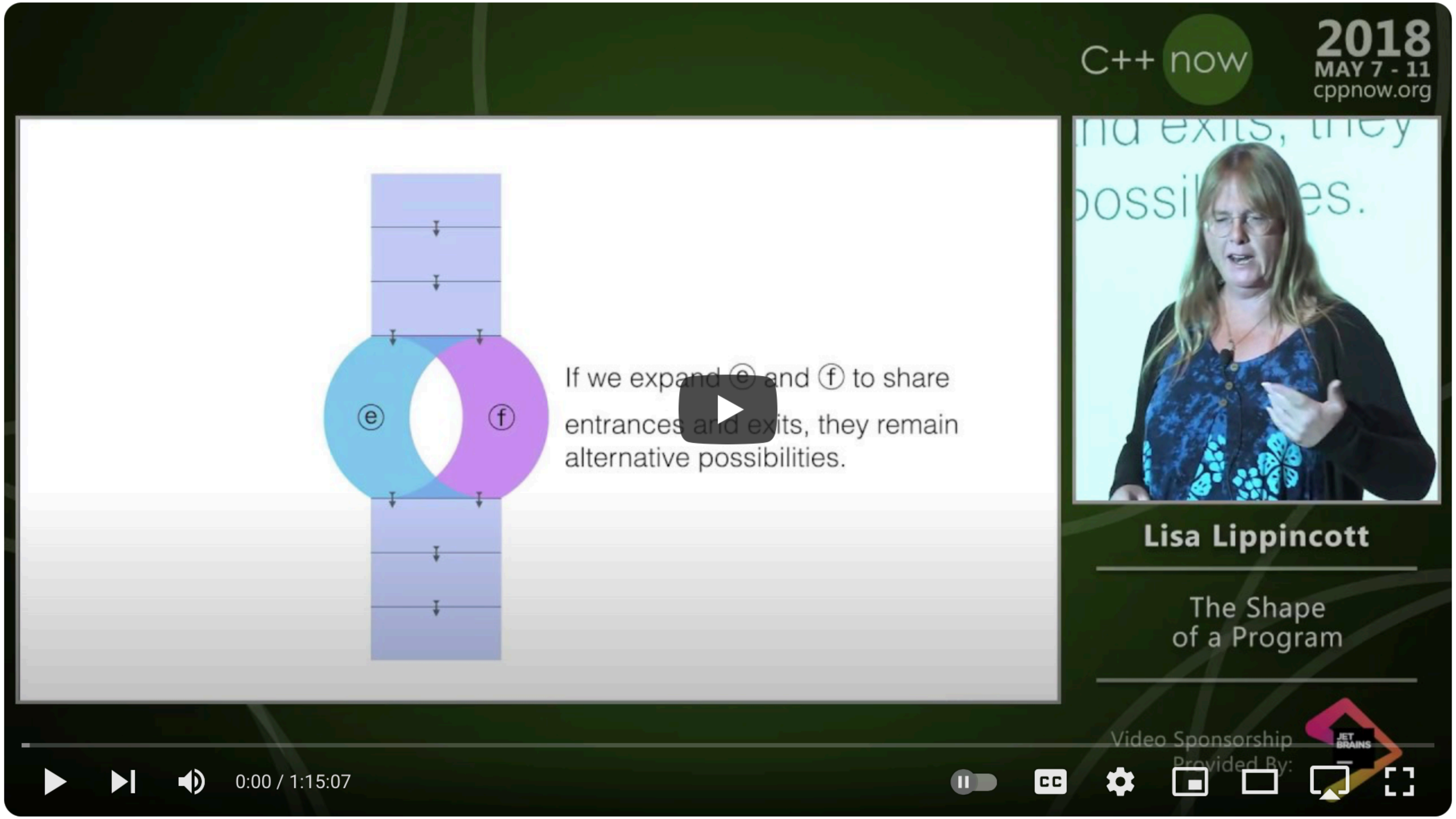
🐘 @ciura_victor@hachyderm.io

🦋 @ciuravictor.bsky.social

## Agenda

3 stories of (a)symmetry

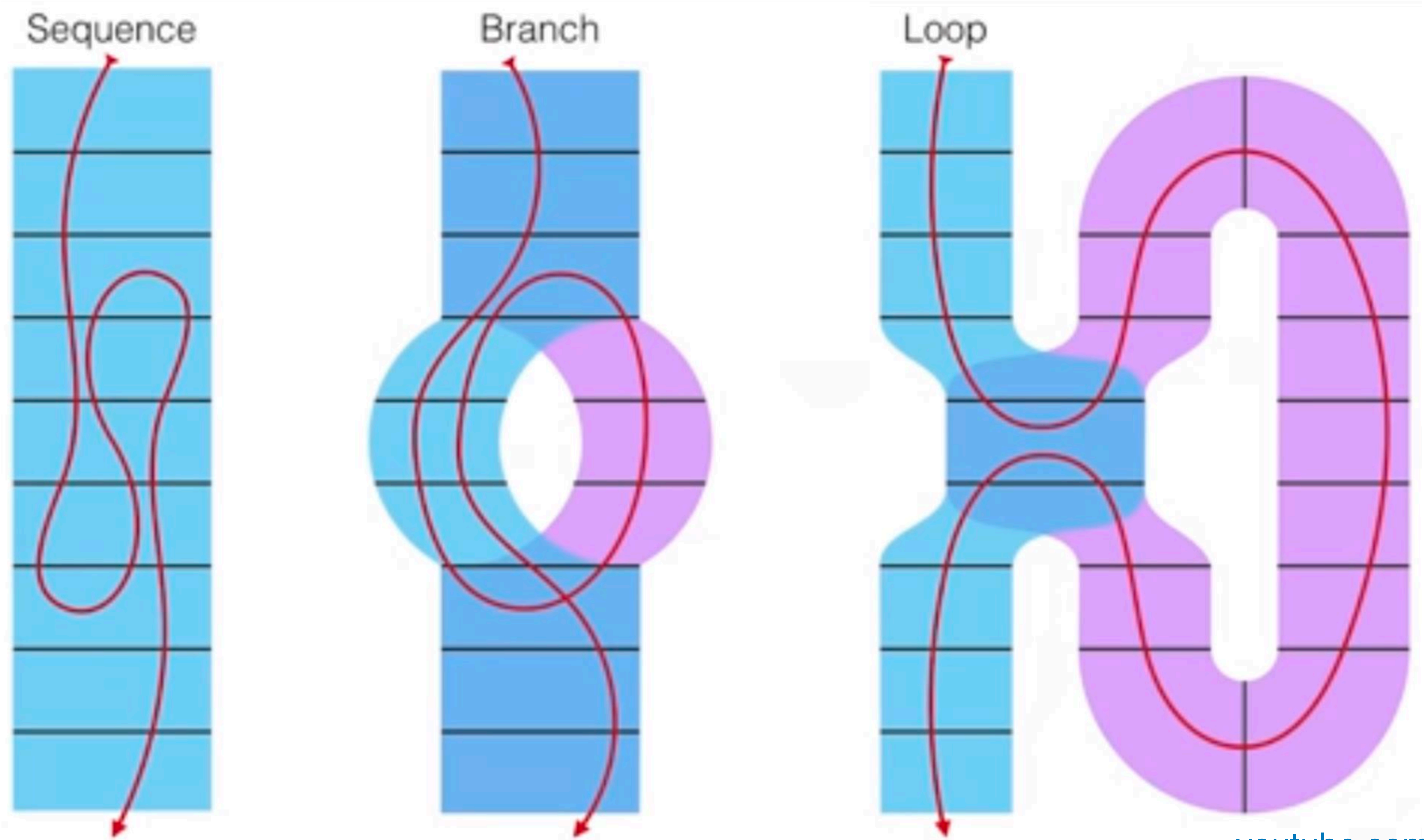
## The Shape of A Program



If we expand ⓔ and ⓕ to share entrances and exits, they remain alternative possibilities.

youtube.com/watch?v=QFIOE1jKv30

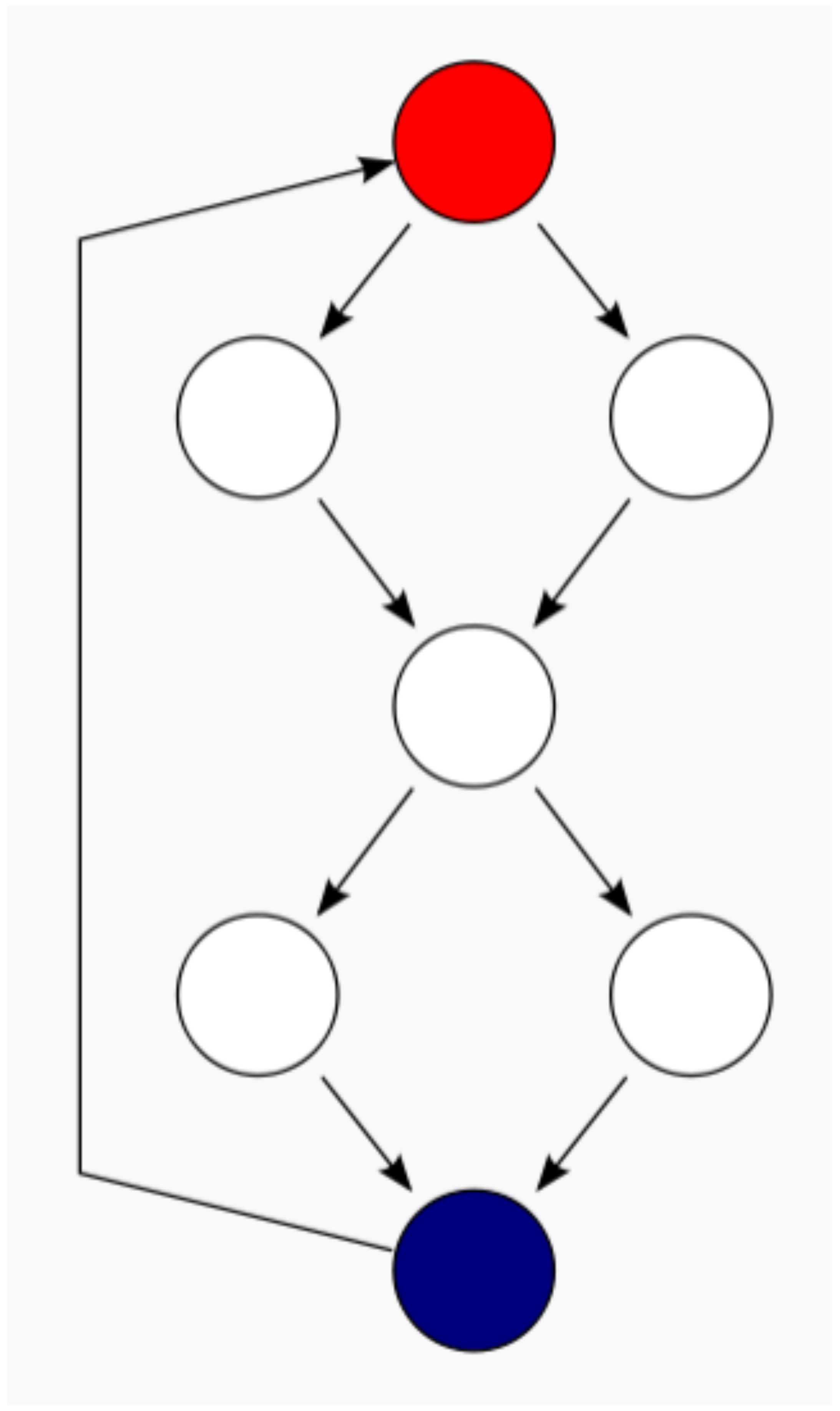
## The Shape of A Program



youtube.com/watch?v=QFIOE1jKv30

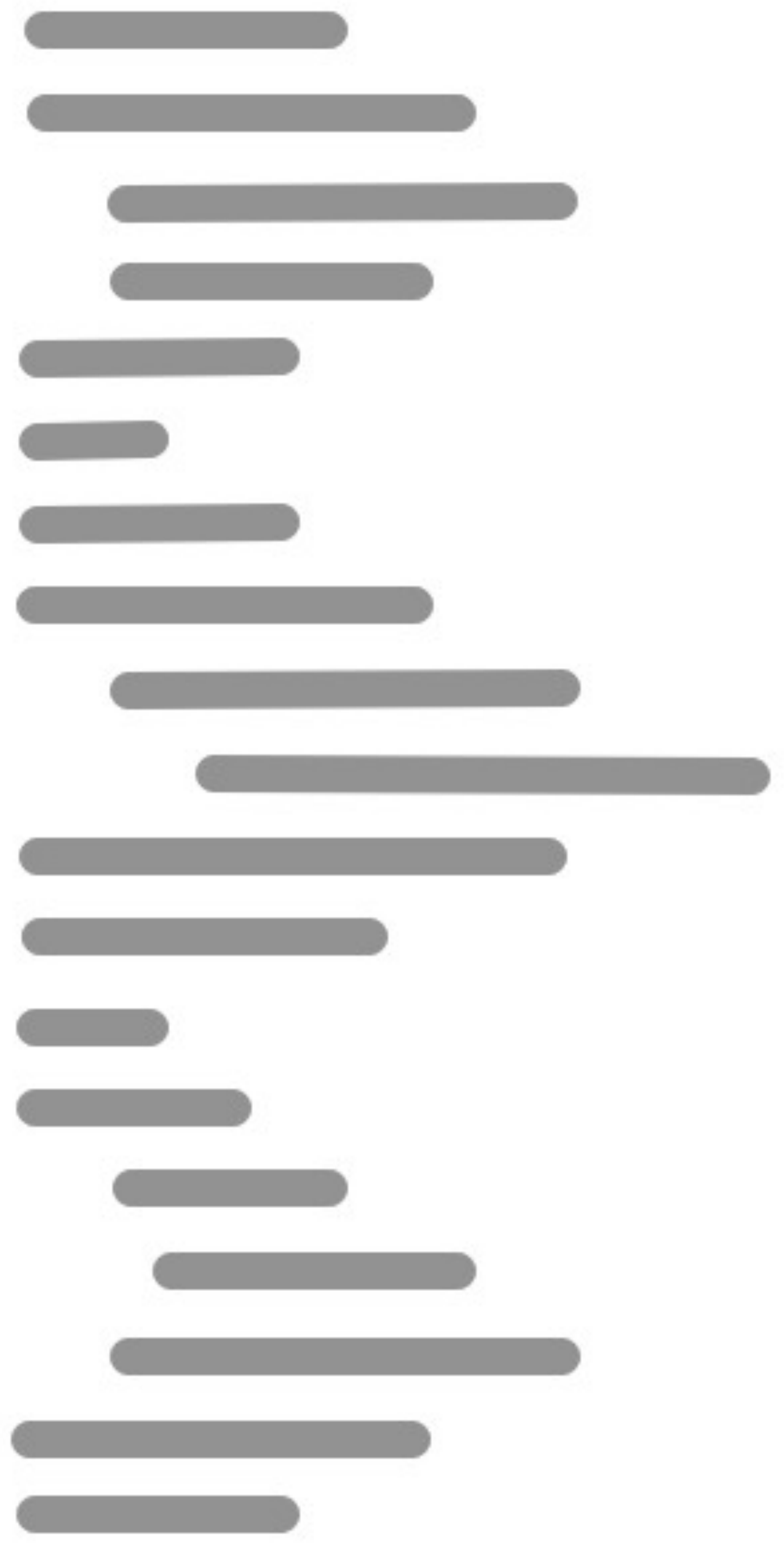
## Cyclomatic Complexity

```
int func()
{
  if (c1())
    f1();
  else
    f2();

  if (c2())
    f3();
  else
    f4();
}
```



wikipedia.org/wiki/Cyclomatic_complexity

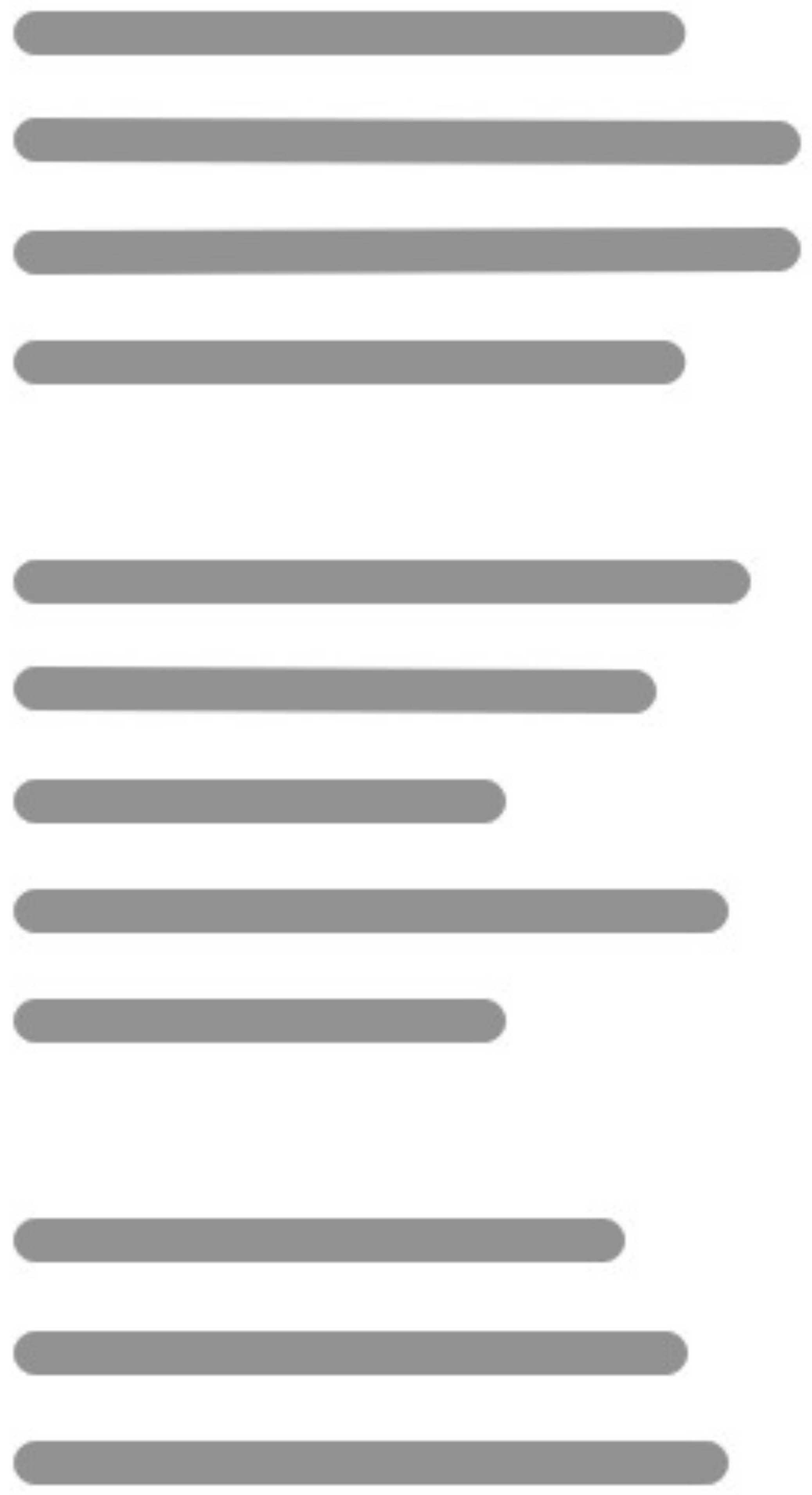
## Shapes of Code



**The saw**

fluentcpp.com/2020/01/14/the-shapes-of-code/

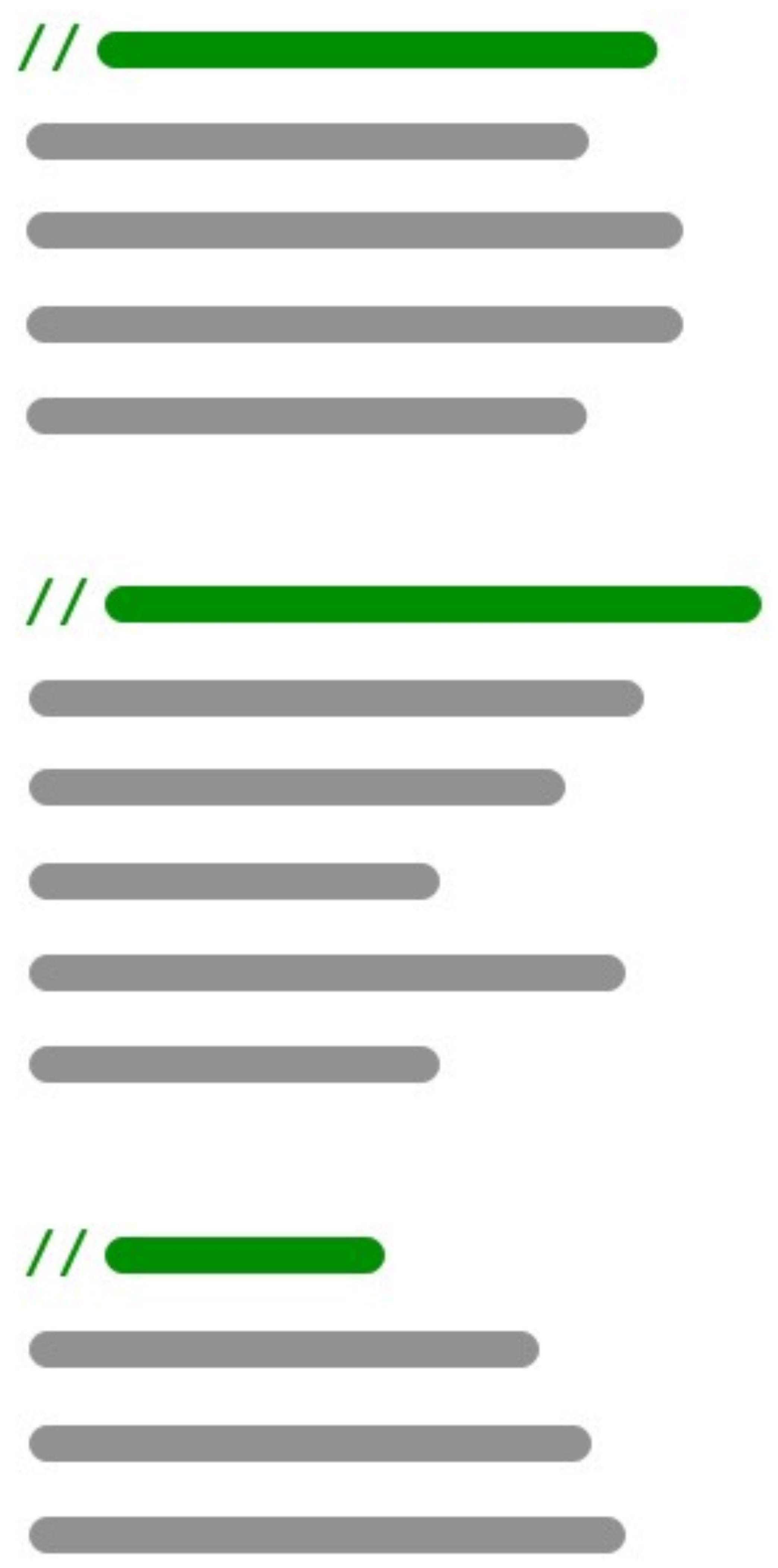
## Shapes of Code



**The paragraphs**

fluentcpp.com/2020/01/14/the-shapes-of-code/

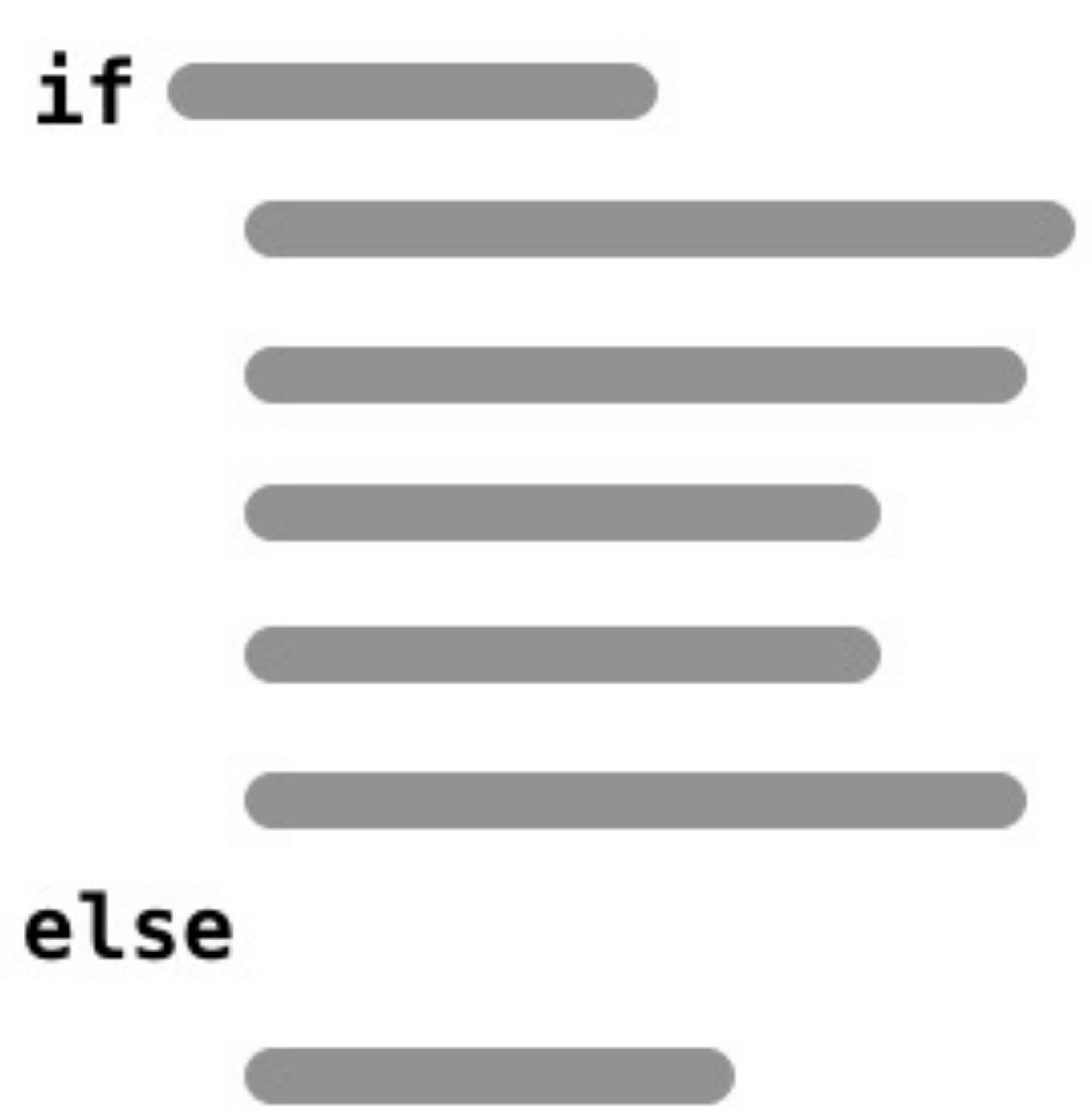
## Shapes of Code

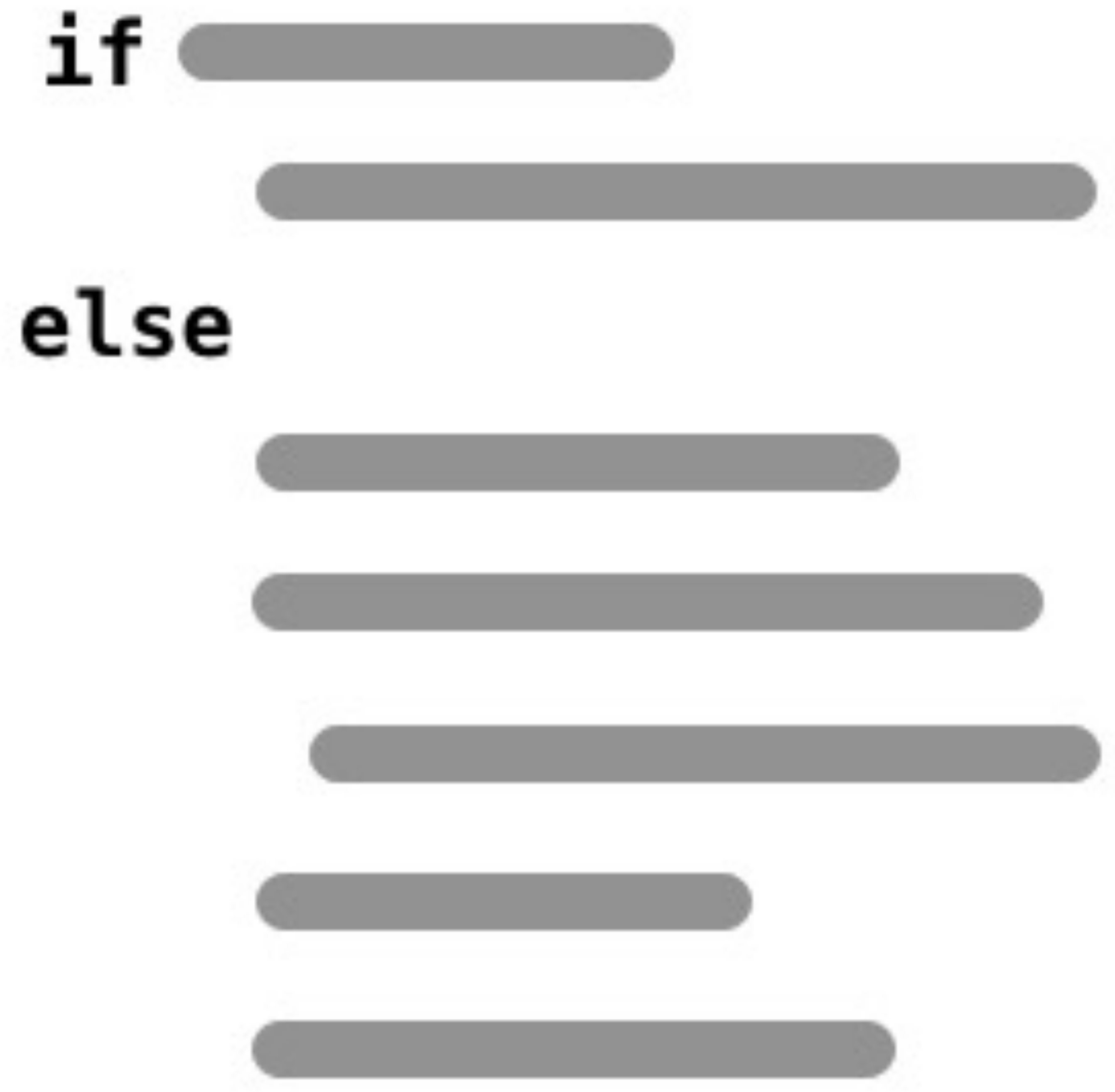


**The paragraphs with headers**

fluentcpp.com/2020/01/14/the-shapes-of-code/

## Shapes of Code





**The unbalanced `if` blocks**

fluentcpp.com/2020/01/14/the-shapes-of-code/

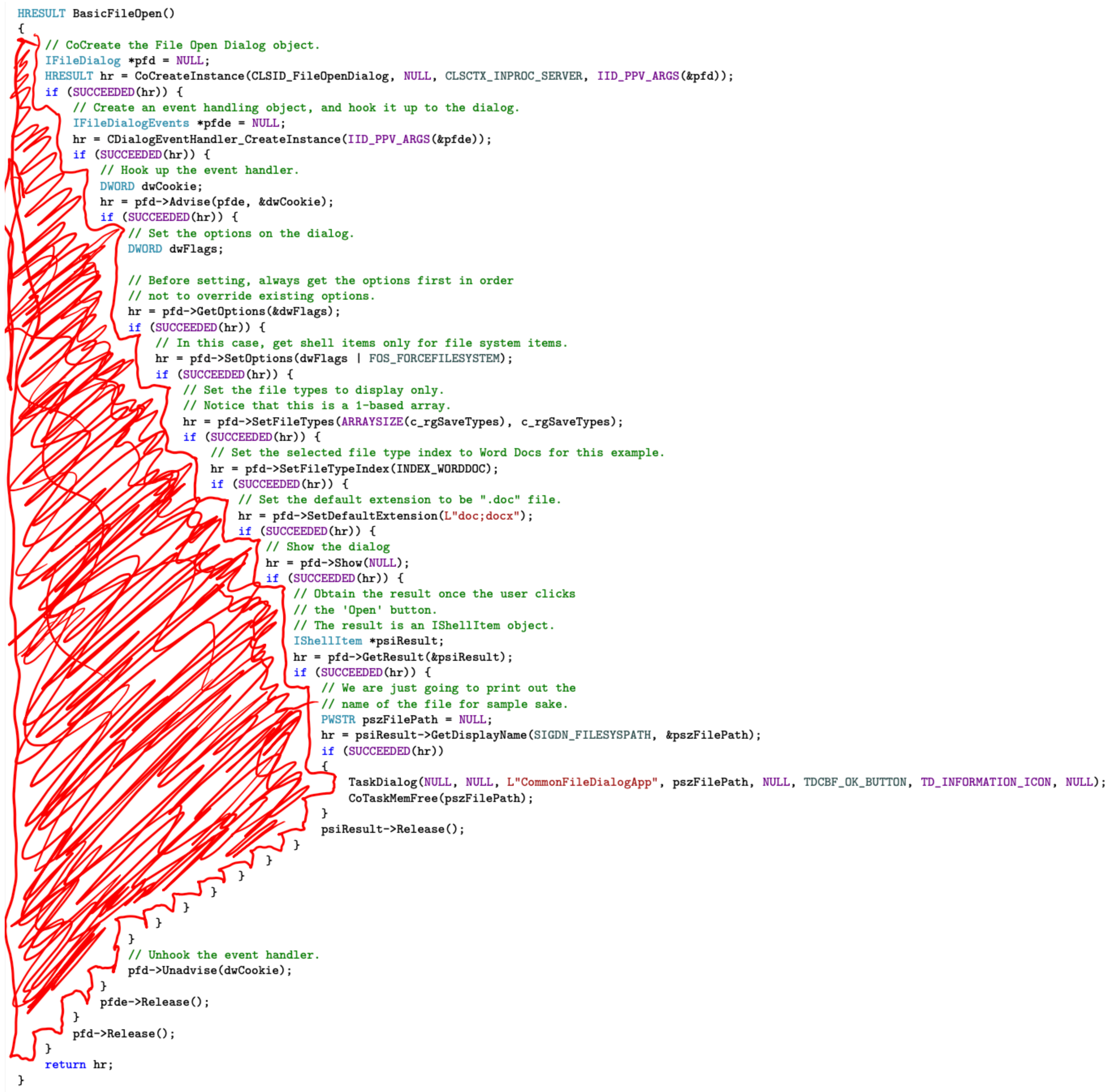
## The Shape of A Program

youtube.com/watch?v=P2IxGnbDkDI

# Program Complexity ?

```cpp
int main()
{
    // Seed with a real random value, if available
    std::random_device r;

    // Choose a random mean between 1 and 6
    std::default_random_engine e1(r());
    std::uniform_int_distribution<int> uniform_dist(1, 6);
    int mean = uniform_dist(e1);
    std::cout << "Randomly-chosen mean: " << mean << '\n';

    // Generate a normal distribution around that mean
    std::seed_seq seed2{r(), r(), r(), r(), r(), r(), r(), r()};
    std::mt19937 e2(seed2);
    std::normal_distribution<> normal_dist(mean, 2);

    std::map<int, int> hist;
    for (int n = 0; n < 10000; ++n) {
        ++hist[std::round(normal_dist(e2))];
    }
    std::cout << "Normal distribution around " << mean << ":\n";
    for (auto p : hist) {
        std::cout << std::fixed << std::setprecision(1)
            << std::setw(2) << p.first << ' ' <<
            std::string(p.second/200, '*') << '\n';
    }
}
```

# Program Complexity ?



```
HRESULT BasicFileOpen()
{
    // CoCreate the File Open Dialog object.
    IFileDialog *pfd = NULL;
    HRESULT hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pfd));
    if (SUCCEEDED(hr)) {
        // Create an event handling object, and hook it up to the dialog.
        IFileDialogEvents *pfde = NULL;
        hr = CDialogEventHandler_CreateInstance(IID_PPV_ARGS(&pfde));
        if (SUCCEEDED(hr)) {
            // Hook up the event handler.
            DWORD dwCookie;
            hr = pfd->Advise(pfde, &dwCookie);
            if (SUCCEEDED(hr)) {
                // Set the options on the dialog.
                DWORD dwFlags;

                // Before setting, always get the options first in order
                // not to override existing options.
                hr = pfd->GetOptions(&dwFlags);
                if (SUCCEEDED(hr)) {
                    // In this case, get shell items only for file system items.
                    hr = pfd->SetOptions(dwFlags | FOS_FORCEFILESYSTEM);
                    if (SUCCEEDED(hr)) {
                        // Set the file types to display only.
                        // Notice that this is a 1-based array.
                        hr = pfd->SetFileTypes(ARRAYSIZE(c_rgSaveTypes), c_rgSaveTypes);
                        if (SUCCEEDED(hr)) {
                            // Set the selected file type index to Word Docs for this example.
                            hr = pfd->SetFileTypeIndex(INDEX_WORDDOC);
                            if (SUCCEEDED(hr)) {
                                // Set the default extension to be ".doc" file.
                                hr = pfd->SetDefaultExtension(L"doc;docx");
                                if (SUCCEEDED(hr)) {
                                    // Show the dialog
                                    hr = pfd->Show(NULL);
                                    if (SUCCEEDED(hr)) {
                                        // Obtain the result once the user clicks
                                        // the 'Open' button.
                                        // The result is an IShellItem object.
                                        IShellItem *psiResult;
                                        hr = pfd->GetResult(&psiResult);
                                        if (SUCCEEDED(hr)) {
                                            // We are just going to print out the
                                            // name of the file for sample sake.
                                            PWSTR pszFilePath = NULL;
                                            hr = psiResult->GetDisplayName(SIGDN_FILESYSPATH, &pszFilePath);
                                            if (SUCCEEDED(hr))
                                            {
                                                TaskDialog(NULL, NULL, L"CommonFileDialogApp", pszFilePath, NULL, TDCBF_OK_BUTTON, TD_INFORMATION_ICON, NULL);
                                                CoTaskMemFree(pszFilePath);
                                            }
                                            psiResult->Release();
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                // Unhook the event handler.
                pfd->Unadvise(dwCookie);
            }
            pfde->Release();
        }
        pfd->Release();
    }
    return hr;
}
```

# Reduce Complexity

```cpp
void DoThing(int index)
{
    if (IsValidIndexOfOtherThing(index))
    {
        if (CanDoSomethingWithNumber(index))
        {
            if (CheckSomethingCriticalAboutValue(index))
            {
                for (auto const& value : GetData(index))
                {
                    switch (value % 3)
                    {
                    case 0:
                        PrintFoo(value);
                        break;

                    case 1:
                        PrintBar(value);
                        break;

                    case 2:
                        PrintBaz(value);
                        break;
                    }
                }
            }
        }
    }
}
```

➡️

```cpp
void DoThing(int index)
{
    if (!IsValidIndexOfOtherThing(index))
    {
        return;
    }

    if (!CanDoSomethingWithNumber(index))
    {
        return;
    }

    if (!CheckSomethingVeryCriticalAboutValue(index))
    {
        return;
    }

    for (auto const& value : GetValuesSimilarTo(index))
    {
        ProcessValue(value);
    }
}
```

Flatten, using **guards**

## Guard Pattern

```
/// e.g., "my_key: 123"
pub fn key_num<'a>(item: &'a str) → Result<(&'a str, i32) > {
  if let Some((key, val)) = item split_once(':') {
    if let Ok(val) = val.trim().parse::<i32>() {
      Ok((key, val))
    } else {
      Err(Error::Static("Can't parse integer"))
    }
  } else {
    Err(Error::Static("Invalid format"))
  }
}
```

## Guard Pattern

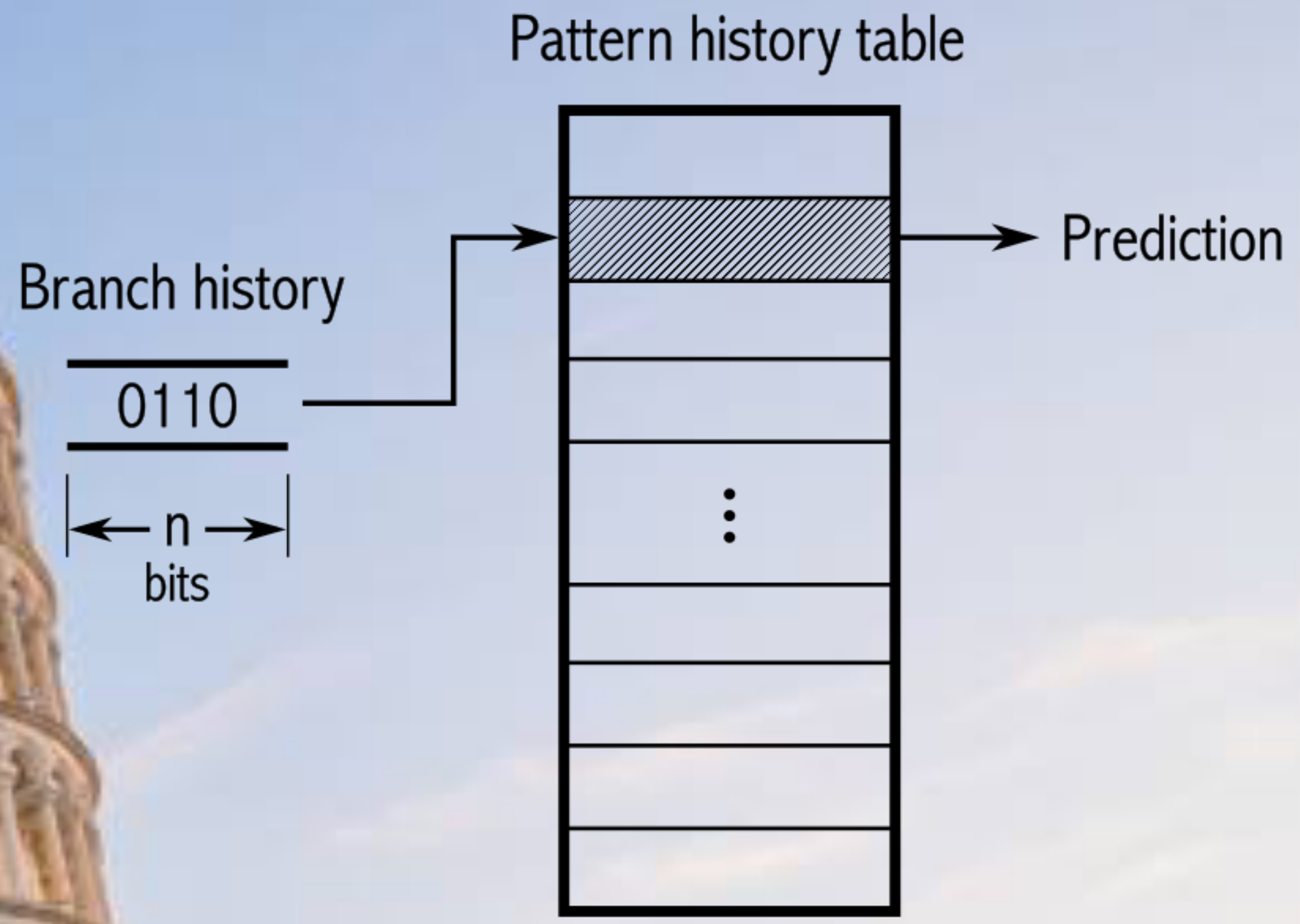
```
/// e.g., "my_key: 123"
pub fn key_num<'a>(item: &'a str) → Result<(&'a str, i32) > {
  let Some((key, val)) = item split_once(':') else {
    return Err(Error::Static("Invalid format"));
  };

  let Ok(val) = val.trim().parse::<i32>() else {
    return Err(Error::Static("Can't parse integer"));
  };

→ Ok((key, val))
}
```

## Guard Pattern

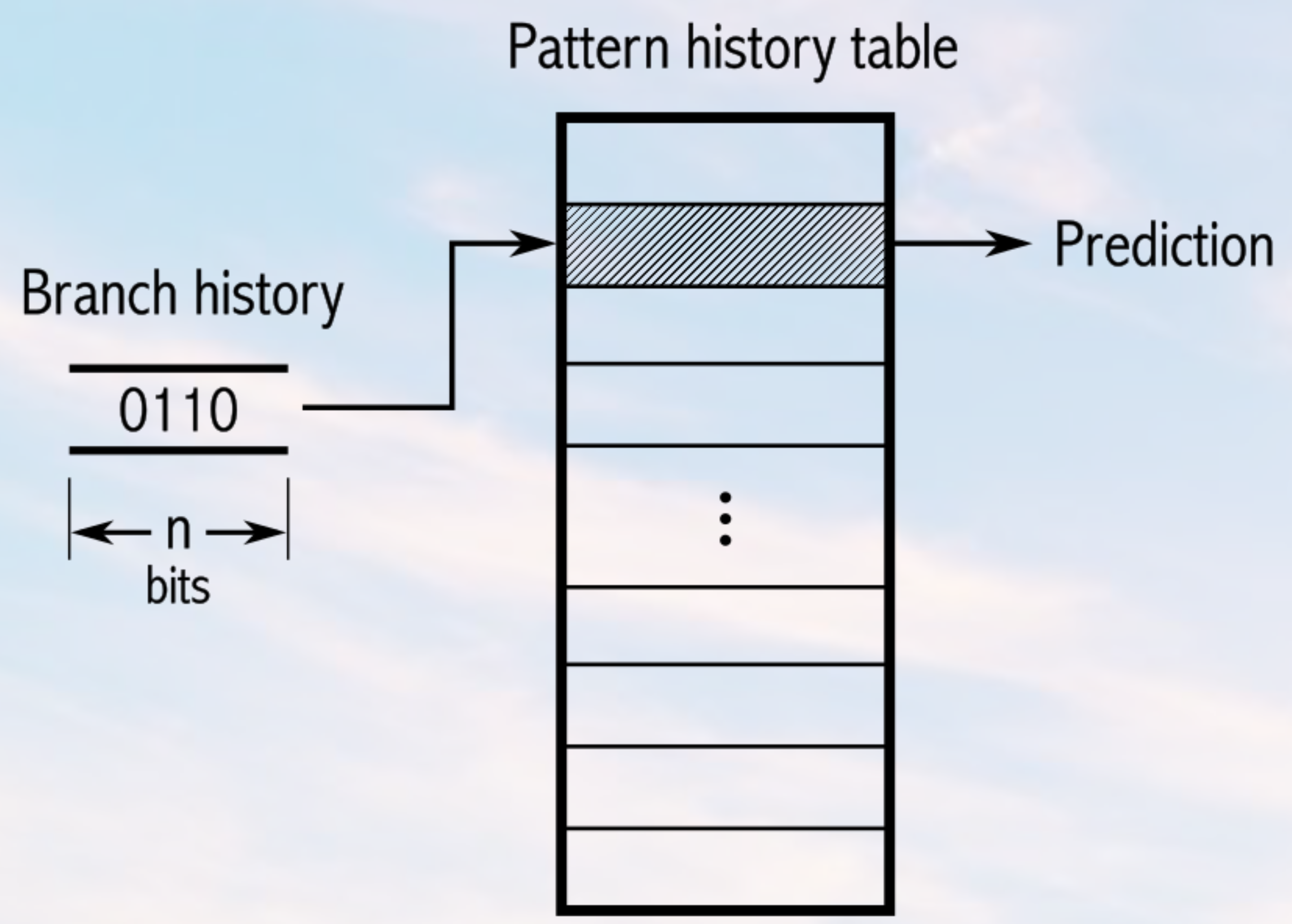
```swift
func getMeaningOfLife() -> Int? {
  42
}

func printMeaningOfLife() {
  if let name = getMeaningOfLife() {
    print(name)
  }
}
```

## Guard Pattern

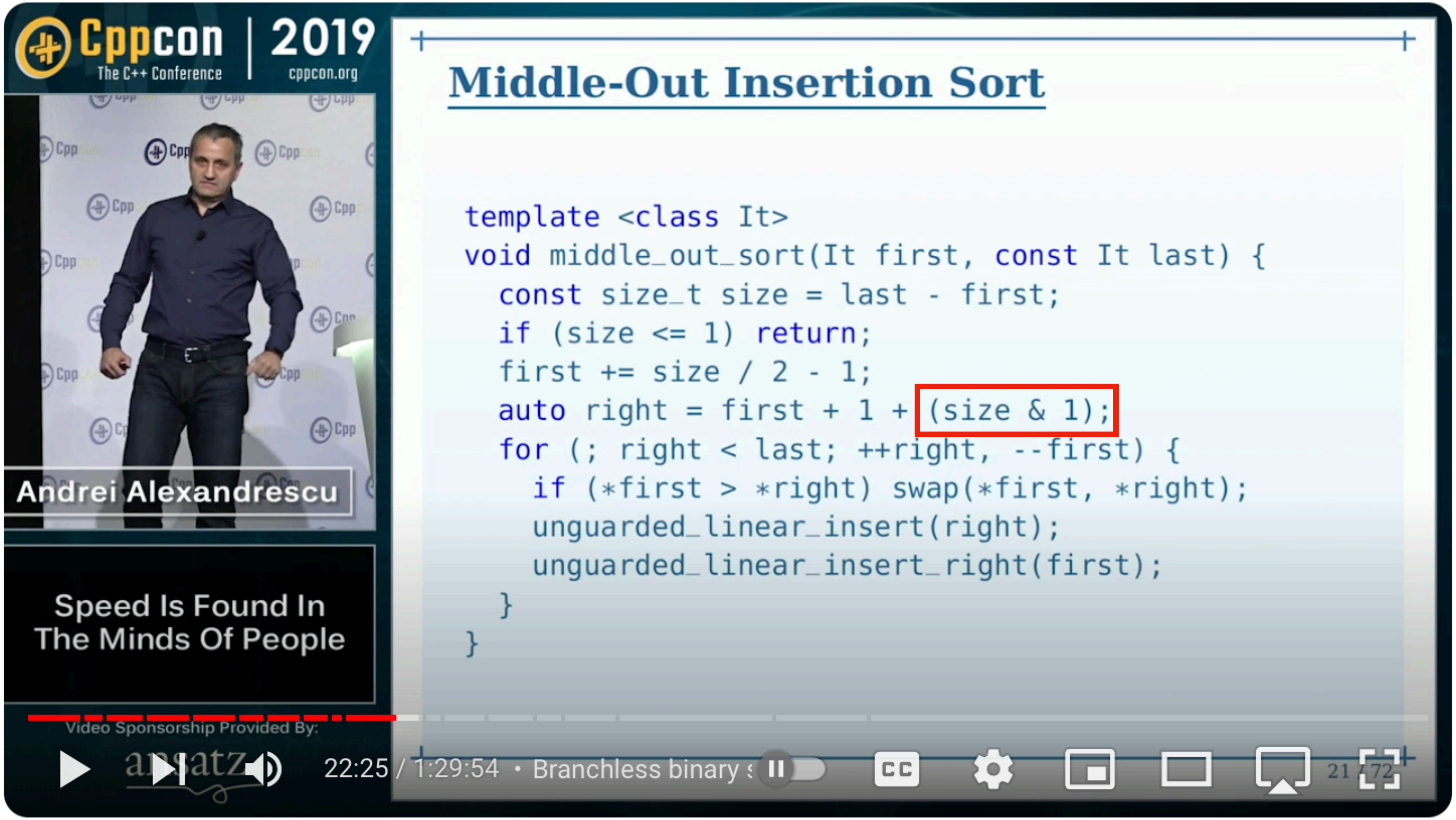
```swift
func printMeaningOfLife() {
  guard let name = getMeaningOfLife() else {
    return
  }

  print(name)
}
```


Pattern history table
Branch history
0110
n
bits
Prediction

## Code that is left-leaning is fast



## Left Leaning ⬅️



Cppcon The C++ Conference | 2019 cppcon.org
Andrei Alexandrescu
Speed Is Found In The Minds Of People
Video Sponsorship Provided By:
22:25 / 1:29:54 • Branchless binary s
21 / 72

**Middle-Out Insertion Sort**

```
template <class It>
void middle_out_sort(It first, const It last) {
  const size_t size = last - first;
  if (size <= 1) return;
  first += size / 2 - 1;
  auto right = first + 1 + (size & 1);
  for (; right < last; ++right, --first) {
    if (*first > *right) swap(*first, *right);
    unguarded_linear_insert(right);
    unguarded_linear_insert_right(first);
  }
}
```

youtube.com/watch?v=FJJTYQYB1JQ

## Left Leaning ⬅️

**“Code that is left-leaning is fast”**

**- Andrei Alexandrescu**

```
auto right = first + 1 + (size & 1);
```

🚫
```
if (size & 1) right++;
```

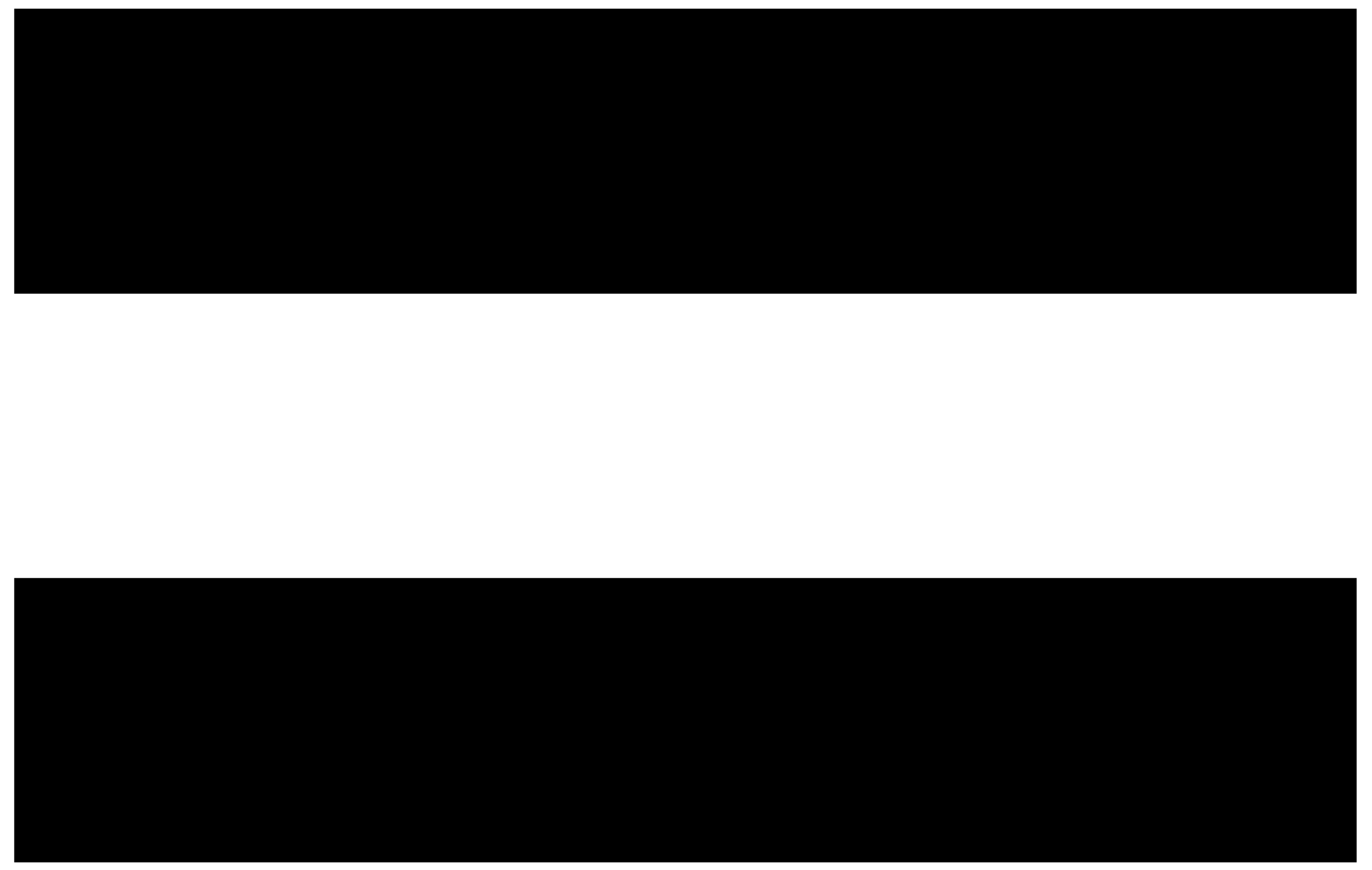
Position in the middle of the array - but differently **if** we have odd or even number of elements.

But there is no `if` statement!

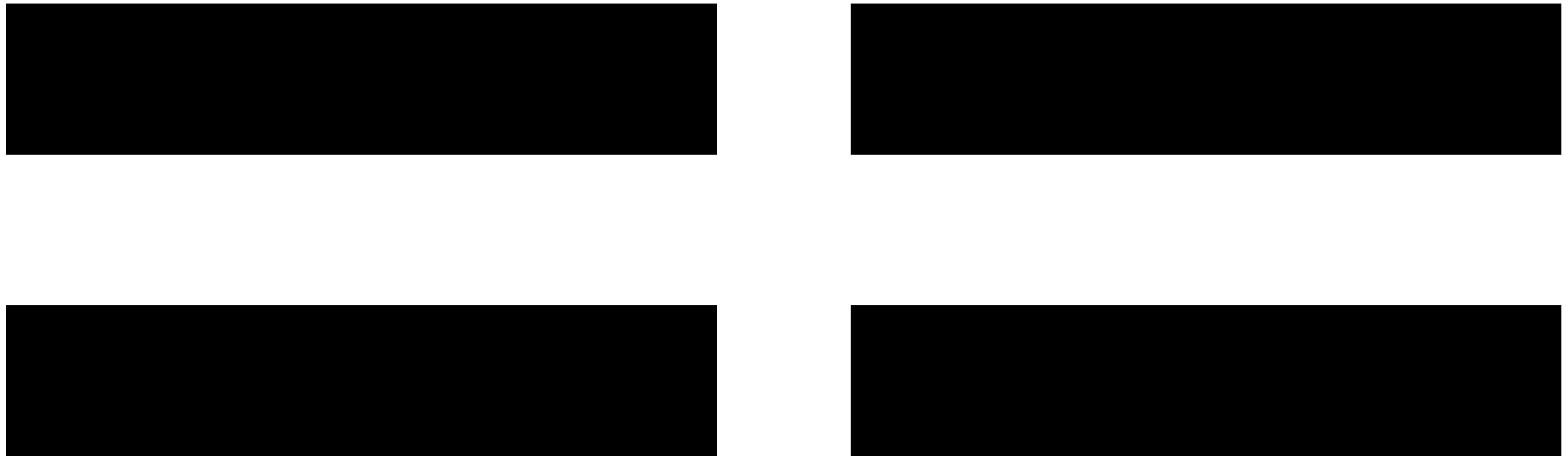
Integrating the conditional within the arithmetic, to avoid branching - no jumps!

## SUPER ASYMMETRY ?

## Is this symmetrical?



## Is this symmetrical?



## Is this symmetrical?

`<=>`

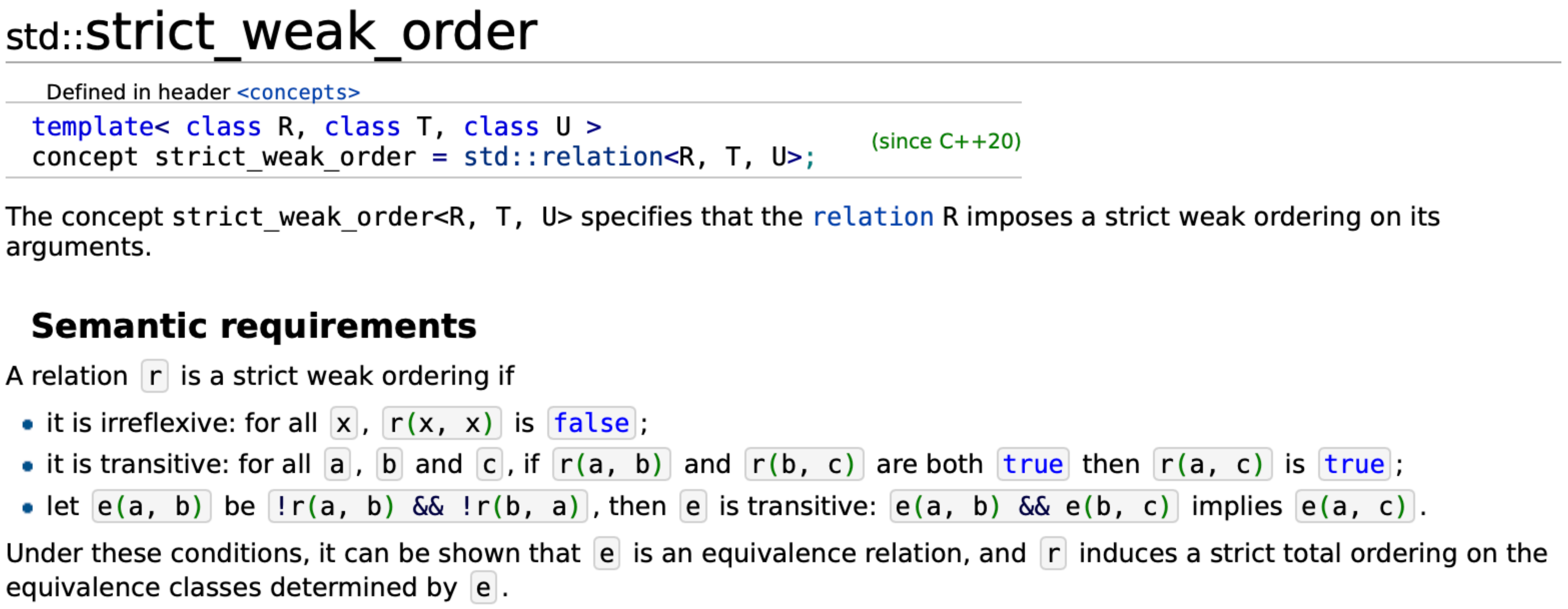
# Strict weak ordering

| | |
|---|---|
| Irreflexivity | `∀ a, comp(a,a)==false` |
| Antisymmetry | `∀ a, b, if comp(a,b)==true => comp(b,a)==false` |
| Transitivity | `∀ a, b, c, if comp(a,b)==true and comp(b,c)==true => comp(a,c)==true` |
| Transitivity of equivalence | `∀ a, b, c, if equiv(a,b)==true and equiv(b,c)==true => equiv(a,c)==true` |

where:

```
equiv(a,b) : comp(a,b)==false && comp(b,a)==false
```

## Concept: Strict weak ordering



std::**strict_weak_order**

Defined in header `<concepts>`

```
template< class R, class T, class U >
concept strict_weak_order = std::relation<R, T, U>;
```
(since C++20)

The concept `strict_weak_order<R, T, U>` specifies that the `relation` R imposes a strict weak ordering on its arguments.

**Semantic requirements**

A relation `r` is a strict weak ordering if

- it is irreflexive: for all `x`, `r(x, x)` is `false`;
- it is transitive: for all `a`, `b` and `c`, if `r(a, b)` and `r(b, c)` are both `true` then `r(a, c)` is `true`;
- let `e(a, b)` be `!r(a, b) && !r(b, a)`, then `e` is transitive: `e(a, b) && e(b, c)` implies `e(a, c)`.

Under these conditions, it can be shown that `e` is an equivalence relation, and `r` induces a strict total ordering on the equivalence classes determined by `e`.

cppreference.com/w/cpp/concepts/strict_weak_order

## EqualityComparable

cppreference.com/w/cpp/named_req/EqualityComparable

| | |
|---|---|
| Reflexivity | `∀ a, (a == a)==true` |
| Symmetry | `∀ a, b, if (a == b)==true => (b == a)==true` |
| Transitivity | `∀ a, b, c, if (a == b)==true and (b == c)==true => (a == c)==true` |

The type must work with `operator==` and the result should have ***standard semantics***.

wikipedia.org/wiki/Equivalence_relation

## Concept: EqualityComparable

cppreference.com/w/cpp/concepts/equality_comparable

```
template< class T, class U >
concept __WeaklyEqualityComparableWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t == u } -> boolean-testable;
    { t != u } -> boolean-testable;
    { u == t } -> boolean-testable;
    { u != t } -> boolean-testable;
  };

template< class T >
concept equality_comparable = __WeaklyEqualityComparableWith<T, T>;
```

wikipedia.org/wiki/Equivalence_relation

# Regular

```
SemiRegular {
          DefaultConstructible, MoveConstructible, CopyConstructible
          MoveAssignable, CopyAssignable, Swappable
          Destructible
}
          +
          EqualityComparable
```

(aka "Stepanov Regular")

## Concept: Regular

```
template <class T>
concept regular = std::semiregular<T> &&
                  std::equality_comparable<T>;


template< class T, class U >
concept __WeaklyEqualityComparableWith =
  requires(const std::remove_reference_t<T>& t,
           const std::remove_reference_t<U>& u) {
    { t == u } -> boolean-testable;
    { t != u } -> boolean-testable;
    { u == t } -> boolean-testable;
    { u != t } -> boolean-testable;
  };

template< class T >
concept equality_comparable = __WeaklyEqualityComparableWith<T, T>;
```

cppreference.com/w/cpp/concepts/regular

Defining **equality** for types is hard 🤕

## #define Equality

**Stepanov** proposes the following *definition*:

" Two objects are **equal** if their corresponding *parts* are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

😓

**stepanovpapers.com/DeSt98.pdf**

## #define Equality

“although it still leaves room for judgement”

**Stepanov** proposes the following *definition*:

" Two objects are **equal** if their corresponding *parts* are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

😓

stepanovpapers.com/DeSt98.pdf

## Three-way comparison

C++20

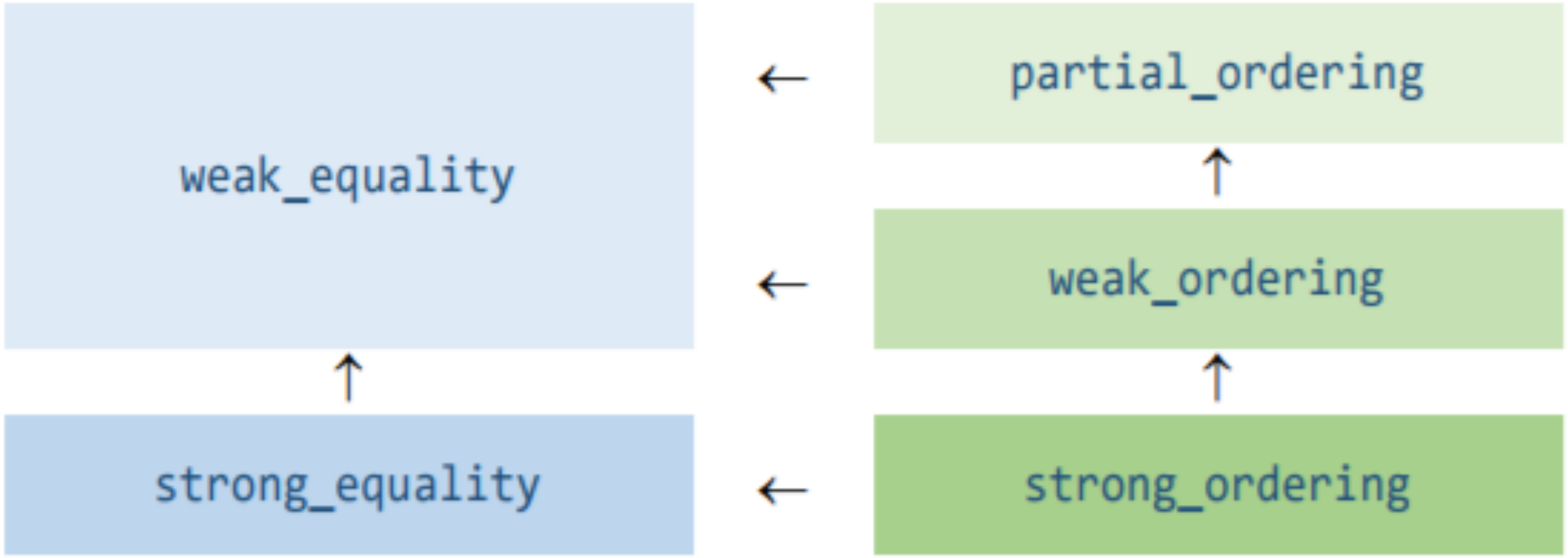
**Bringing consistent comparison operations...**

```
operator <=>
```

```
(a <=> b) <  0  if  a < b
(a <=> b) >  0  if  a > b
(a <=> b) == 0  if  a and b are equal/equivalent
```

## Three-way comparison

**The comparison categories for:** `operator <=>`



**It's all about *relation strength***

## Three-way comparison

<, <=, >, >= synthesized from operator<=>
!=  synthesized from operator==

```
operator<=>

operator!=

operator==
```

Convenience

Efficiency ?

The problem: implement <=> optimally for "wrapper" types

```
struct S {
  vector<string> names;
  auto operator<=>(S const&) const = default;
};
```

wg21.link/P1185

## <=>

```
template<typename T>
strong_ordering operator<=>(vector<T> const& lhs, vector<T> const& rhs)
{
    size_t min_size = min(lhs.size(), rhs.size());

    for (size_t i = 0; i != min_size; ++i)
    {
        if (auto const cmp = compare_3way(lhs[i], rhs[i]); cmp != 0) {
            return cmp;
        }
    }

    return lhs.size() <=> rhs.size();
}
```

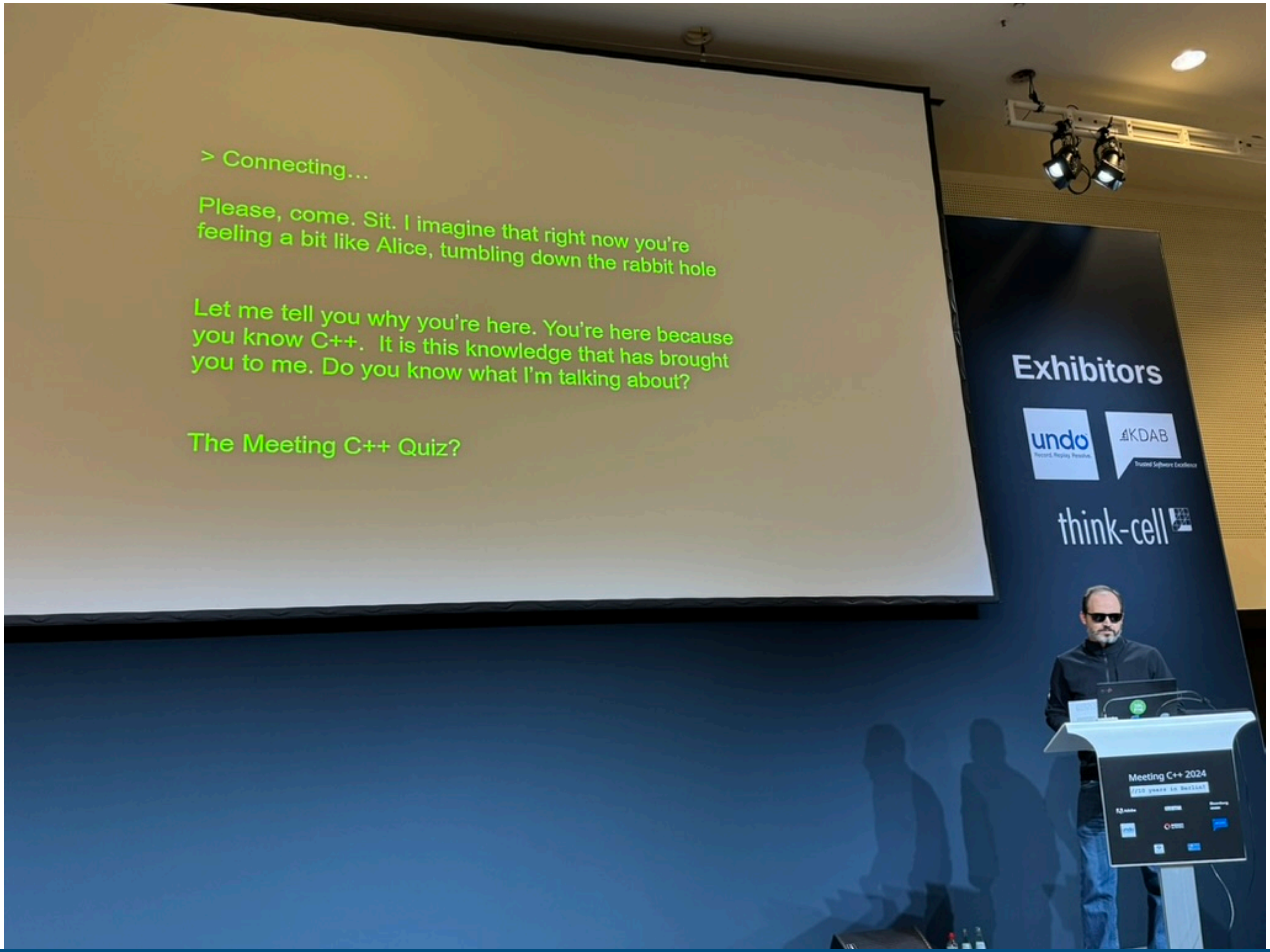
## <=>

```
template<typename T>
bool operator==(vector<T> const& lhs, vector<T> const& rhs)
{
    // short-circuit on size early
    const size_t size = lhs.size();
    if (size != rhs.size()) {
        return false;
    }

    for (size_t i = 0; i != size; ++i) {
        // use ==, not <=>, in all nested comparisons
        if (lhs[i] != rhs[i]) {
            return false;
        }
    }

    return true;
}
```

## 🙄 Real life code is much simpler & clear than this nonsense!

🙄 **Real life code is much simpler & clear than this nonsense!**

# Quiz Time



> Connecting...

Please, come. Sit. I imagine that right now you're feeling a bit like Alice, tumbling down the rabbit hole

Let me tell you why you're here. You're here because you know C++. It is this knowledge that has brought you to me. Do you know what I'm talking about?

The Meeting C++ Quiz?

## Move fast & break things

**/std:c++20**

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

Names have been changed to protect the innocent 😁

## Move fast & break things

/std:c++20

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

  🔥operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

Names have been changed to protect the innocent 😁

## Move fast & break things

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

## Move fast & break things

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

  //operator const char*() const;
};
```

```
  MyString str1("Hello");
//if ("Hello" == str1)
  if (str1 == "Hello")
    puts("equal");
  else
    puts("NOT equal");
```

## Move fast & break things

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

## Move fast & break things

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
  //bool operator!=(const char* str) const;

    operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

## Let's go back...

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

## Let's go back...

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    explicit operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

## Let's go back...

```
...
    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    explicit operator const char*() const;
};
```

## Let's go back...

```
...
    bool operator==(const char* str) const;
    bool operator!=(const char* str) const;

    explicit operator const char*() const;
};
```

```
ERROR: 'bool MyString::operator ==(const char *) const': rewritten candidate
function was excluded from overload resolution because a corresponding operator!=
declared in the same scope

could be 'bool MyString::operator ==(const char *) const'
[synthesized expression 'y == x']

'bool MyString::operator ==(const char *) const': rewritten candidate function was
excluded from overload resolution because a corresponding operator!= declared in
the same scope
or       'built-in C++ operator==(const char [6], const char [6])'
'==': cannot convert argument 2 from 'MyString' to 'const char [6]'
or       'built-in C++ operator==(const char *, const char *)'
'==': cannot convert argument 2 from 'MyString' to 'const char *'
```

## C++ 20 Equality

This is expected after the compilers implemented P2468R2

- The Equality Operator You Are Looking For (2022)
- *"This paper details some changes to make rewriting equality in expressions less of a breaking change"* 😁

## C++ 20 Equality

This is expected after the compilers implemented P2468R2

- The Equality Operator You Are Looking For (2022)
- *"This paper details some changes to make rewriting equality in expressions less of a breaking change"* 😁

In C++20, the presence of the operator!= instructs the compiler to *suppress* operator rewriting since the intention is to keep C++17 code compiling.

In C++17 mode, the code would have produced exactly the same result.

## C++ 20 Equality

To fix the issue, consider making the operator const char*() explicit

and ~~removing~~ the operator!= to make the code more C++20 friendly

## C++ 20 Equality

```
class MyString
{
private:
    char* m_Data;

public:
    MyString(const char* str);
    ~MyString();

    bool operator==(const char* str) const;
    ~~bool operator!=(const char* str) const;~~
    explicit operator const char*() const;
};
```

```
MyString str1("Hello");
if ("Hello" == str1)
  puts("equal");
else
  puts("NOT equal");
```

# Ship it!

## Be Deliberate

**Incidental vs. deliberate symmetry**

## Should We Care?

We should be looking to identify patterns in code,
to see when such constructs exhibit some sort of symmetry
that is advantageous in some way for:

- reliability
- performance
- maintenance/extensibility
- discoverability

## One more thing (++)

Incrementing variables in for-loops:

`i++`

- overused
- nonsensical
- imbalanced

`i-=-1`

- hipster
- expressive
- symmetric

credit: *probably* Ólafur Waage

# Symmetry in Code
## ¿Should We Care?

**Meeting C++**

November 2024

🐦 @ciura_victor
🐘 @ciura_victor@hachyderm.io
🦋 @ciuravictor.bsky.social

**Victor Ciura**
Principal Engineer
Rust Tooling @ Microsoft