C++ 🎓 UNIverse

Meeting C++

August 5, 2021

@ciura_victor

Victor Ciura
Principal Engineer

CAPHYON

Abstract

Performance has always been the goal for C++ and that can frequently come in conflict with teachability. Since I was a student, twenty years ago, until today C++ has been a staple diet in universities across the globe. But “C++ as a first language”... really?

There is a lot of room for us to make C++ more teachable and improve the quality of C++ teaching in UNI, so long as we’re not talking about CS1.

First, students have to get over the hurdle of being algorithmic thinkers and then we can give them a language that has these sharp edges.

Is this a lost cause? I think not. Modern C++ is simpler and safer and we have numerous opportunities to make it more teachable at the same time.
"The king is dead, long live the king!"

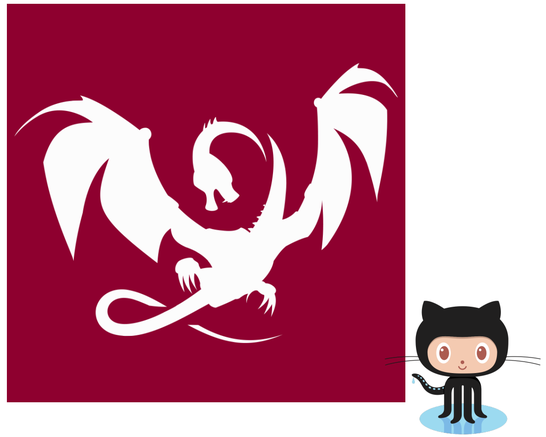
Free discussion (not recorded)

hopin

hopin.com/events/mini-conference-with-victor-ciura

About me

Advanced Installer



Clang Power Tools

Free/OSS

@ciura_victor


My C++ UNIverse

University of Craiova

UCV - Computer Engineering Department

My 🎓 UNIverse

I’m a regular guest at the *Computer Engineering Department* of my Alma Mater, University of Craiova, where I give invited lectures & workshops on using C++, STL, algorithms, optimization techniques and programming principles.

My 🎓 UNIverse

I’m a regular guest at the *Computer Engineering Department* of my Alma Mater, University of Craiova, where I give invited lectures & workshops on using C++, STL, algorithms, optimization techniques and programming principles.

For 6 years, I gave a series of workshops on
“Using C++/STL for Competitive Programming and Software Development”
(we coached teams for student competitions, eg. ACM ICPC-SEERC)

My 🎓 UNIverse

I'm a regular guest at the *Computer Engineering Department* of my Alma Mater, University of Craiova, where I give invited lectures & workshops on using C++, STL, algorithms, optimization techniques and programming principles.

For 6 years, I gave a series of workshops on
"Using C++/STL for Competitive Programming and Software Development"
(we coached teams for student competitions, eg. ACM ICPC-SEERC)

In June-July every year, in collaboration with my friends in academia, I organize and teach a free workshop: *Open4Tech Summer School for Software*
(college & high-school students)

My 🎓 UNIverse

Topics I covered over the years in my lectures & workshops:

- programming techniques
- algorithms
- graphs & trees
- C++
- functional programming (Haskell/C++)
- hashing algorithms & containers

Student Expectations @ Y1 Sem I

"Software is eating the World"...

... and I want to be a part of it !

Student Expectations @ Y1 Sem I

First Encounter

C++

Powerful as hell. Can actually do anything.

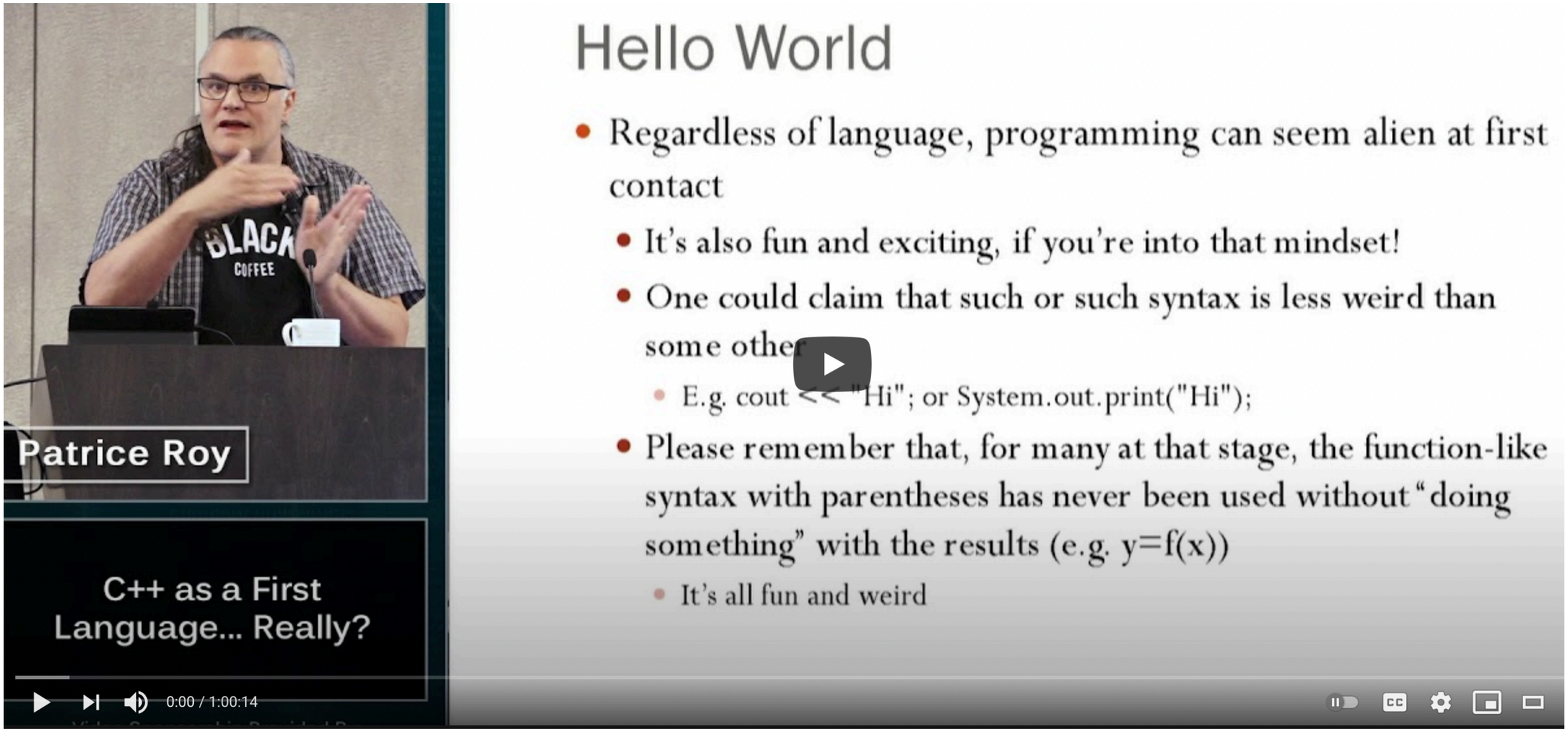
God save you if something goes wrong.

First Encounter

CS 1

C++ as a first language... really?

C++ as a first language... really?



C++ as a First Language... Really? - Patrice Roy - CppCon 2019

https://www.youtube.com/watch?v=AyhPigwhwbk

Common themes I keep hearing (C++ community):

Common themes I keep hearing (C++ community):

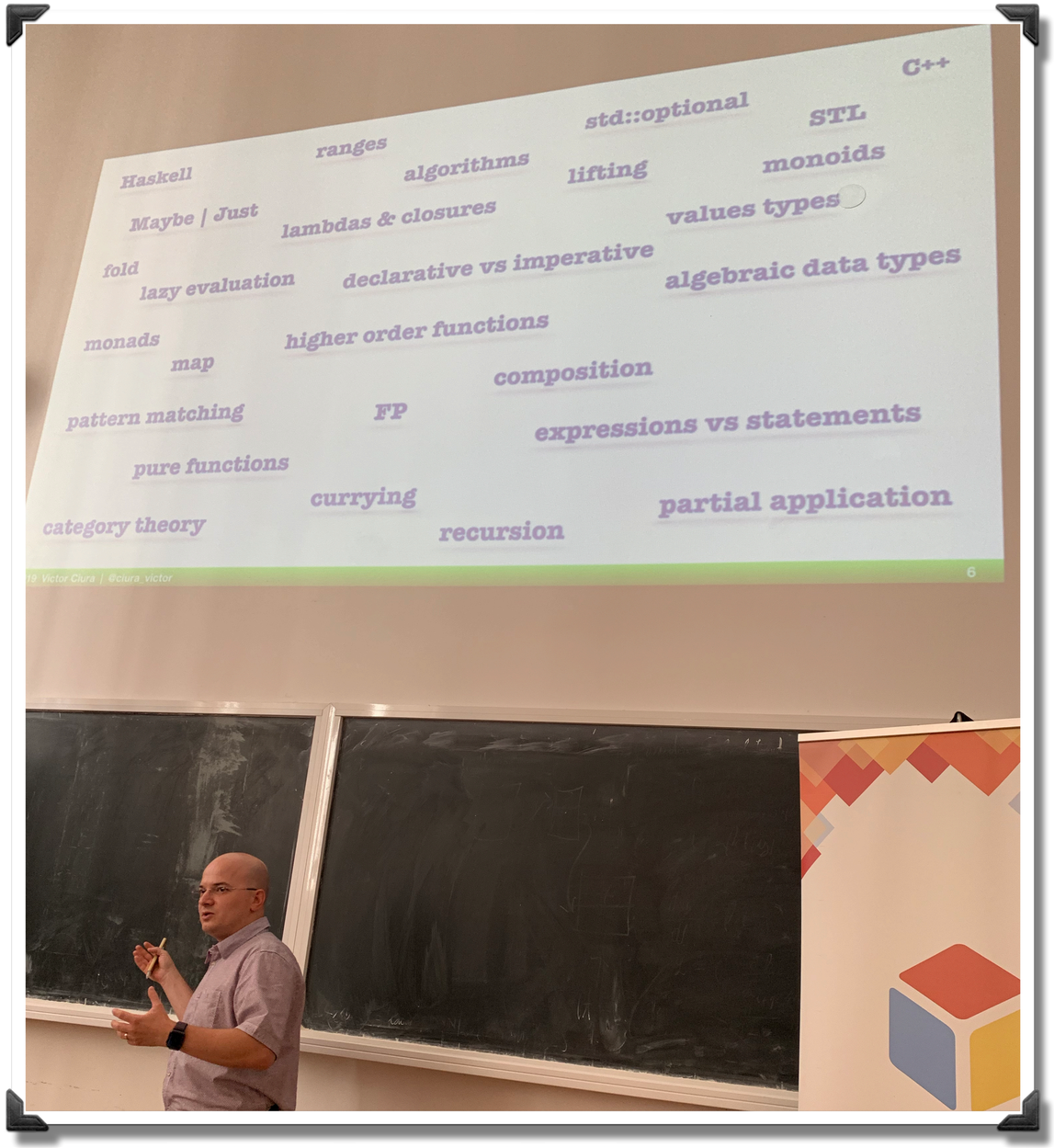
There is a lot of room for us to make C++ more teachable and improve the quality of C++ teaching in UNI, so long as we're not talking about CS1.

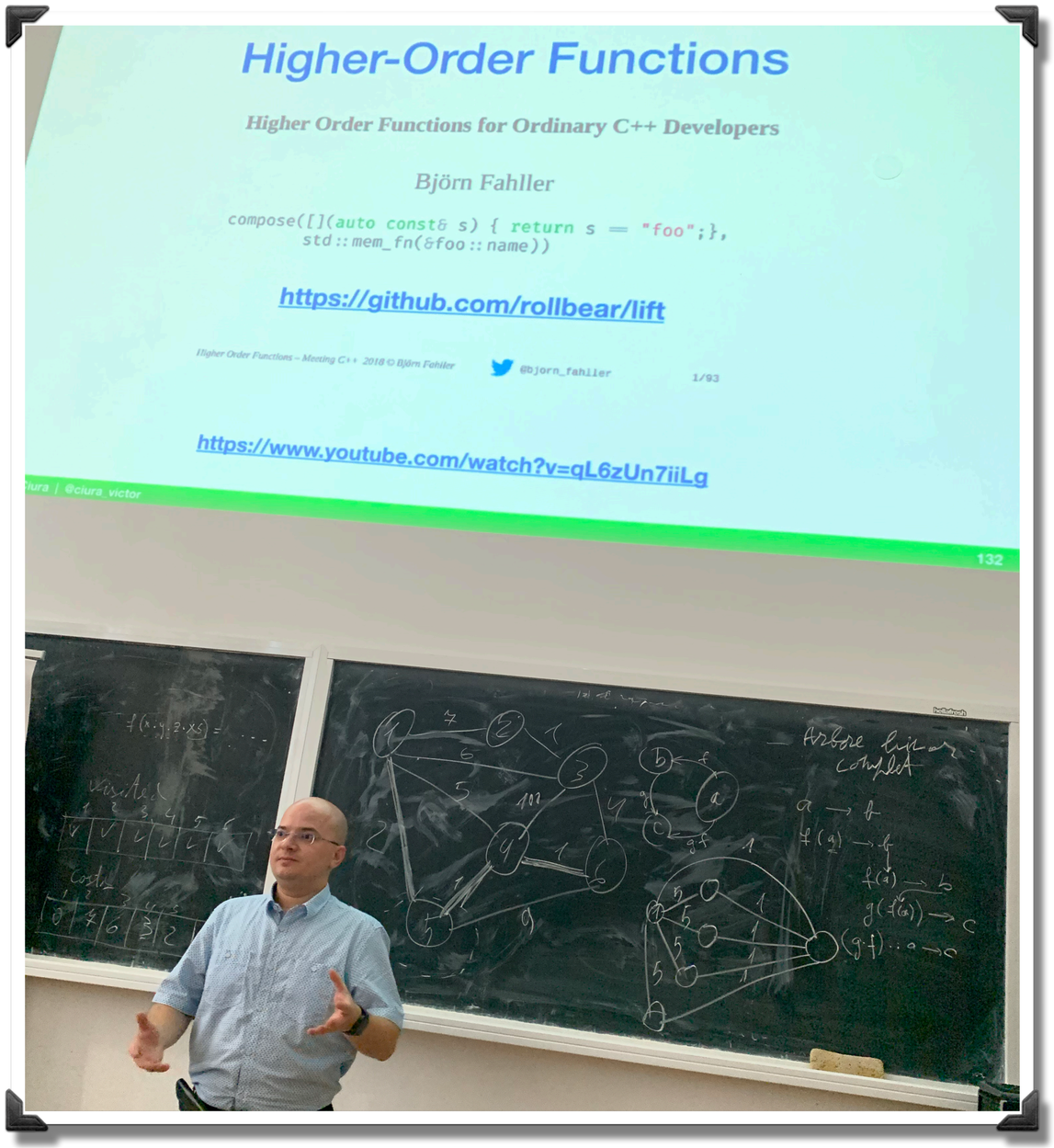
Common themes I keep hearing (C++ community):

There is a lot of room for us to make C++ more teachable and improve the quality of C++ teaching in UNI, so long as we're not talking about CS1.

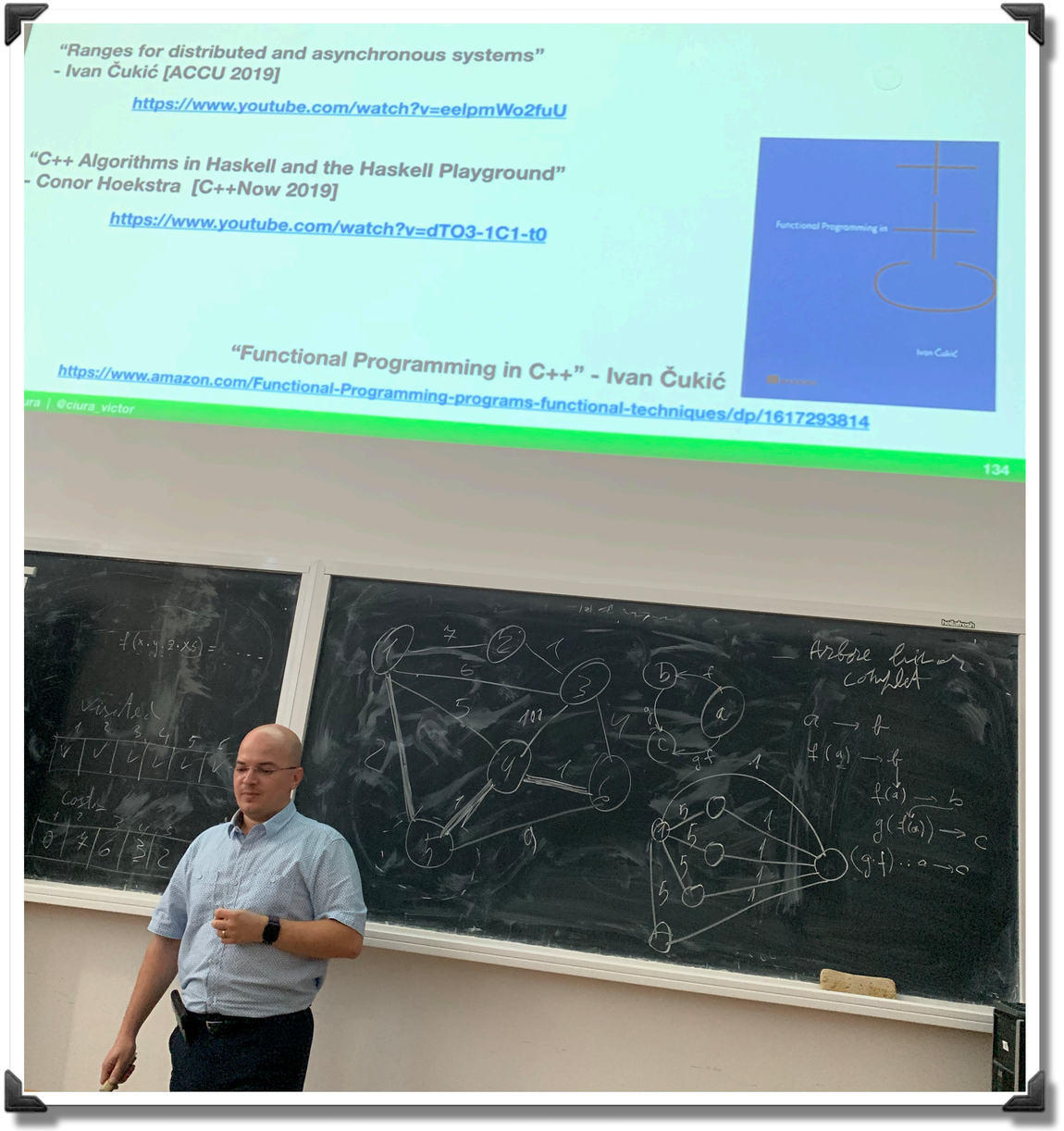
First, students have to get over the hurdle of being algorithmic thinkers and then we can give them a language that has these sharp edges.

Curry On Functional Programming



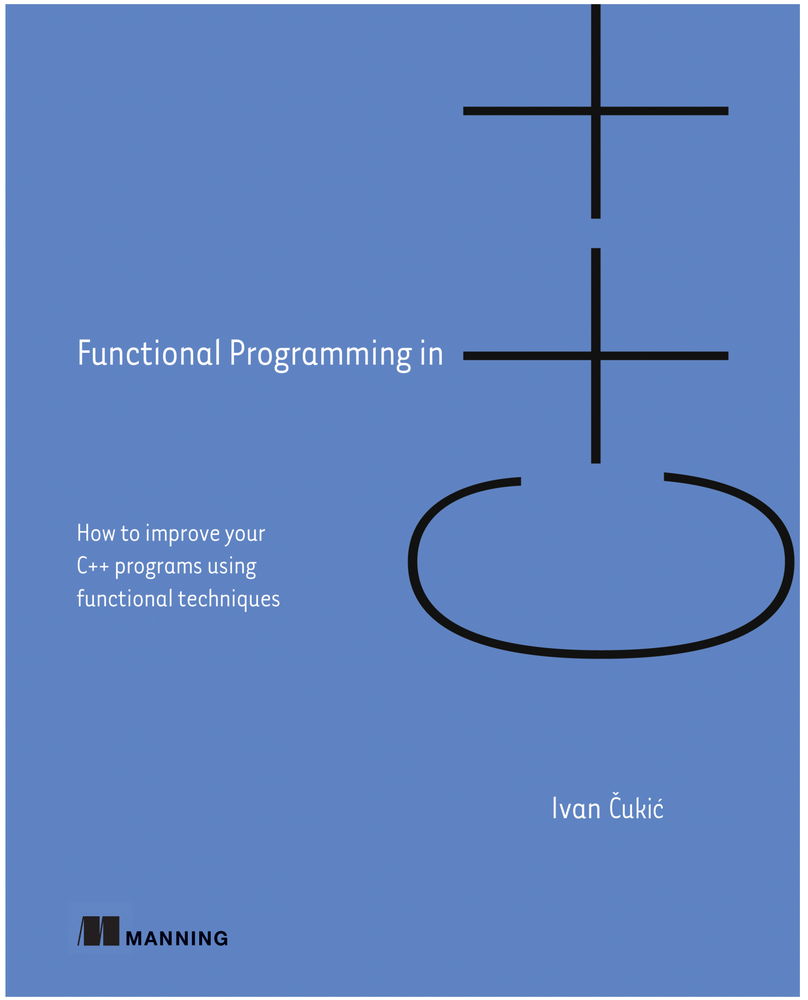


Curry On Functional Programming



Curry On Functional Programming

C++20
ranges



STL Algorithms: Principles & Practice

Sharp edges

Language

Library

```
[](){
}();
```

```
template<typename T>
void func(T && arg)
{
}
```

std::pair<T1,T2>::pair

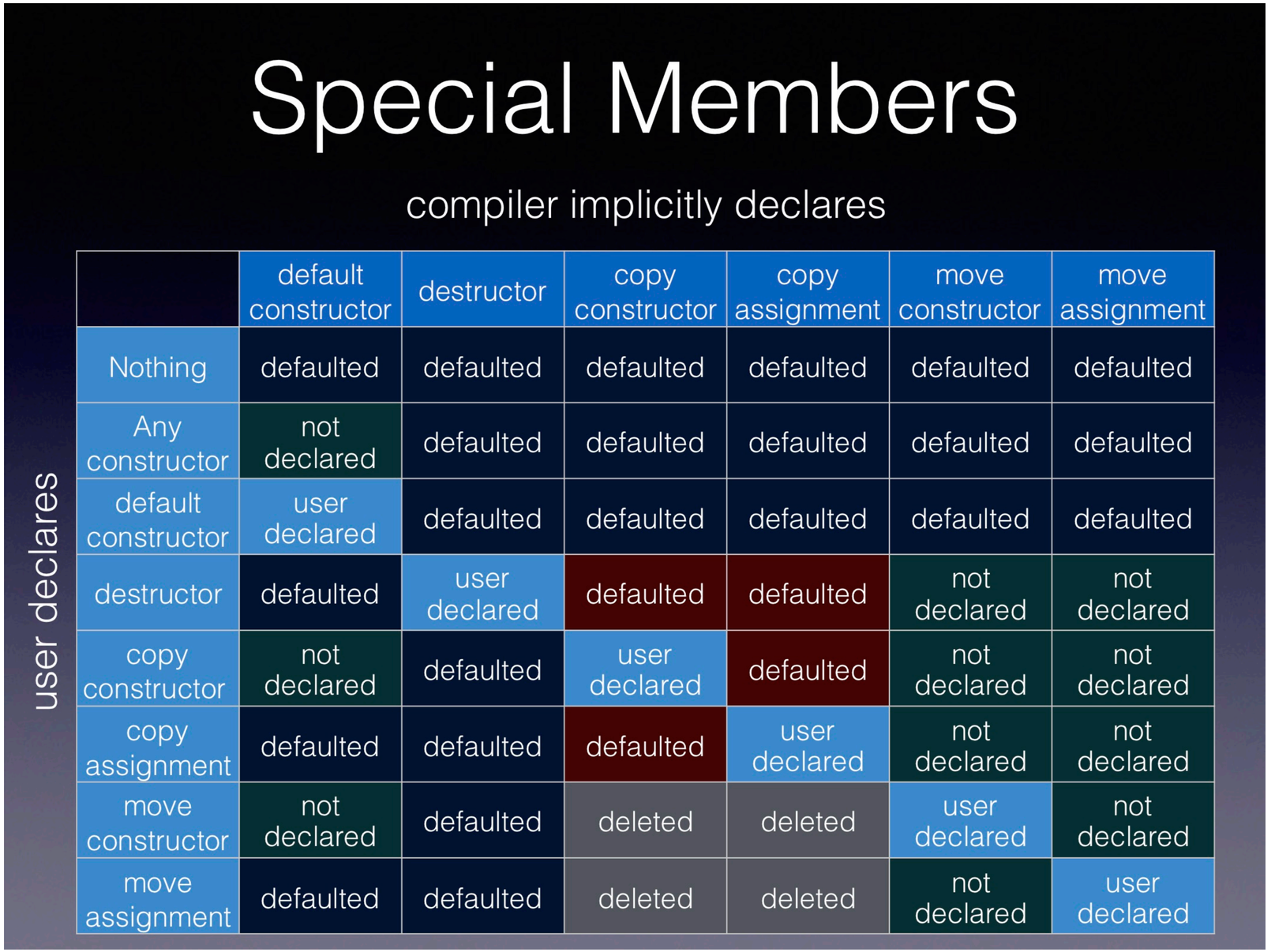
`pair();`	(1)	(until C++11)
`constexpr pair();`		(since C++11) (conditionally explicit)
`pair( const T1& x, const T2& y );`	(2)	(until C++11)
`pair( const T1& x, const T2& y );`		(since C++11) (until C++14) (conditionally explicit)
`constexpr pair( const T1& x, const T2& y );`		(since C++14) (conditionally explicit)
`template< class U1, class U2 >` `pair( U1&& x, U2&& y );`	(3)	(since C++11) (until C++14) (conditionally explicit)
`template< class U1, class U2 >` `constexpr pair( U1&& x, U2&& y );`		(since C++14) (conditionally explicit)
`template< class U1, class U2 >` `pair( const pair<U1, U2>& p );`	(4)	(until C++11)
`template< class U1, class U2 >` `pair( const pair<U1, U2>& p );`		(since C++11) (until C++14) (conditionally explicit)
`template< class U1, class U2 >` `constexpr pair( const pair<U1, U2>& p );`		(since C++14) (conditionally explicit)
`template< class U1, class U2 >` `pair( pair<U1, U2>&& p );`	(5)	(since C++11) (until C++14) (conditionally explicit)
`template< class U1, class U2 >` `constexpr pair( pair<U1, U2>&& p );`		(since C++14) (conditionally explicit)

`template< class... Args1, class... Args2 >`
`pair( std::piecewise_construct_t,`
`std::tuple<Args1...> first_args,`
`std::tuple<Args2...> second_args );`
	(6)	(since C++11)
		(until C++20)

`template< class... Args1, class... Args2 >`
`constexpr pair( std::piecewise_construct_t,`
`std::tuple<Args1...> first_args,`
`std::tuple<Args2...> second_args );`
		(since C++20)

`pair( const pair& p ) = default;`	(7)	
`pair( pair&& p ) = default;`	(8)	(since C++11)

Sharp edges



	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Starting a project

Don't reinvent the wheel,
take on some dependencies

package management

Some examples
that perplex students

C++ Strings

```cpp
const std::string str = "Modern C++";

std::string s1 {"Modern C++", 3};
std::string s2 {str, 3};

std::cout << "S1: " << s1 << "\n";
std::cout << "S2: " << s2 << "\n";
```

output:

```
> S1: Mod
> S2: ern C++
```

twitter.com/vzverovich

C++ Strings

std::string's 11 constructors

```
basic_string();                                                                  (until C++17)
explicit basic_string( const Allocator& alloc );
basic_string() noexcept(noexcept( Allocator() )) :                               (since C++17)
    basic_string( Allocator() ) {}                                               (until C++20)
explicit basic_string( const Allocator& alloc ) noexcept;                (1)
constexpr basic_string() noexcept(noexcept( Allocator() )) :
    basic_string( Allocator() ) {}                                               (since C++20)
explicit constexpr basic_string( const Allocator& alloc ) noexcept;
```

```
basic_string( size_type count,
              CharT ch,                                                          (until C++20)
              const Allocator& alloc = Allocator() );                    (2)
constexpr basic_string( size_type count,                                         (since C++20)
                        CharT ch,
                        const Allocator& alloc = Allocator() );
```

```
basic_string( const basic_string& other,
              size_type pos,                                                     (until C++17)
              size_type count = std::basic_string::npos,
              const Allocator& alloc = Allocator() );
basic_string( const basic_string& other,                                 (3)     (since C++17)
              size_type pos,                                                     (until C++20)
              const Allocator& alloc = Allocator() );
constexpr basic_string( const basic_string& other,                               (since C++20)
                        size_type pos,
                        const Allocator& alloc = Allocator() );
```

```
basic_string( const basic_string& other,
              size_type pos,                                                     (since C++17)
              size_type count,                                                   (until C++20)
              const Allocator& alloc = Allocator() );                    (3)
constexpr basic_string( const basic_string& other,                               (since C++20)
                        size_type pos,
                        size_type count,
                        const Allocator& alloc = Allocator() );
```

```
basic_string( const CharT* s,
              size_type count,                                                   (until C++20)
              const Allocator& alloc = Allocator() );                    (4)
constexpr basic_string( const CharT* s,                                          (since C++20)
                        size_type count,
                        const Allocator& alloc = Allocator() );
```

```
basic_string( const CharT* s,                                                    (until C++20)
              const Allocator& alloc = Allocator() );                    (5)
constexpr basic_string( const CharT* s,                                          (since C++20)
                        const Allocator& alloc = Allocator() );
```

```
template< class InputIt >
basic_string( InputIt first, InputIt last,                                       (until C++20)
              const Allocator& alloc = Allocator() );                    (6)
template< class InputIt >
constexpr basic_string( InputIt first, InputIt last,                             (since C++20)
                        const Allocator& alloc = Allocator() );
```

```
basic_string( const basic_string& other );                               (7)     (until C++20)
constexpr basic_string( const basic_string& other );                             (since C++20)

basic_string( const basic_string& other, const Allocator& alloc );       (7)     (since C++11)
                                                                                 (until C++20)
constexpr basic_string( const basic_string& other, const Allocator& alloc );    (since C++20)

basic_string( basic_string&& other ) noexcept;                           (8)     (since C++11)
                                                                                 (until C++20)
constexpr basic_string( basic_string&& other ) noexcept;                         (since C++20)

basic_string( basic_string&& other, const Allocator& alloc );            (8)     (since C++11)
                                                                                 (until C++20)
constexpr basic_string( basic_string&& other, const Allocator& alloc );          (since C++20)

basic_string( std::initializer_list<CharT> ilist,                                (since C++11)
              const Allocator& alloc = Allocator() );                    (9)     (until C++20)
constexpr basic_string( std::initializer_list<CharT> ilist,                      (since C++20)
                        const Allocator& alloc = Allocator() );

template < class T >                                                             (since C++17)
explicit basic_string( const T& t, const Allocator& alloc = Allocator() );       (until C++20)
template < class T >                                                     (10)
explicit constexpr basic_string( const T& t,                                     (since C++20)
                                 const Allocator& alloc = Allocator() );

template < class T >                                                             (since C++17)
basic_string( const T& t, size_type pos, size_type n,                            (until C++20)
              const Allocator& alloc = Allocator() );                    (11)
template < class T >
constexpr basic_string( const T& t, size_type pos, size_type n,                  (since C++20)
                        const Allocator& alloc = Allocator() );
```

No compiler diagnostics/warnings 😕

C++ Weekly - Ep 262

www.youtube.com/watch?v=3MOw1a9B7kc

Modern C++

Enough string_view to Hang Ourselves ?

It turns out to be ***easy*** to convert **[by design]**

a `std::string` to a `std::string_view`,

or a `std::vector/array` to a `std::span`,

so that dangling is almost the default behavior.

www.youtube.com/watch?v=xwP4YCP_0q0

Modern C++

```
void example()
{
  std::string_view sv = std::string("dangling");


  std::cout << sv;
}
```

Modern C++

```
void example()
{
  std::string_view sv = std::string("dangling");
                      // object backing the pointer will be destroyed
                      // at the end of the full-expression
  std::cout << sv;
}
```

Lifetime with std::string_view (C++17)

std::string_view isn't a drop-in replacement for const std::string&

```
std::string str() {
  return std::string("long_string_helps_to_detect_issues");
}
```

```
const std::string& s = str();
std::cout << s << '\n';
```

lifetime extended
prints the correct result ✓

```
std::string_view sv = str();
std::cout << sv << '\n';
```

lifetime not extended
prints nonsense ✗

const lvalue reference binds to rvalue and provides lifetime extension. But there is no lifetime extension for std:string_view.

👉 For short strings this issue might be hard to detect due to short string optimization (SSO). The problem becomes obvious with longer (dynamically allocated) strings.

@walletfox

CppCoreGuidelines

C++ Strings

Nah, nobody reads docs...

We have tools 🤖

clang-tidy

bugprone-dangling-handle

C++ Strings

`clang -Wlifetime` **Experimental**

`-Wdangling-gsl` diagnosed starting with **Clang 10**

clang-tidy string checks

- abseil-string-find-startswith
- boost-use-to-string
- bugprone-string-constructor
- bugprone-string-integer-assignment
- bugprone-string-literal-with-embedded-nul
- bugprone-suspicious-string-compare
- modernize-raw-string-literal
- performance-faster-string-find
- performance-inefficient-string-concatenation
- readability-redundant-string-cstr
- readability-redundant-string-init
- readability-string-compare

just `string` checks

Order From Chaos...

Students

vs.

`std::sort()`

Order From Chaos...

Students

vs.

`std::sort()`

```
template<class RandomIt, class Compare>
constexpr void sort(RandomIt first, RandomIt last, Compare comp);
```

`Compare` Concept

`Compare` << `BinaryPredicate` << `Predicate` << `FunctionObject` << `Callable`

Why is this one special ?

Because ~50 STL facilities (algorithms & data structures) expect some `Compare` requirement.

https://en.cppreference.com/w/cpp/named_req/Compare

Compare Concept

What are the requirements for a Compare type ?

```
Compare << BinaryPredicate << Predicate << FunctionObject << Callable
```

```
bool comp(*iter1, *iter2);
```

But what kind of *ordering* relationship is needed for the *elements* of the collection ?

https://en.cppreference.com/w/cpp/named_req/Compare

Compare Concept

But what kind of *ordering* relationship is needed 🤔

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`

Their intuition tends to gravitate towards 👉

```
{ Partial ordering }
```

https://en.wikipedia.org/wiki/Partially_ordered_set

Compare Concept

But what kind of *ordering* relationship is needed 🤔

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`

Their intuition tends to gravitate towards 👉

`{ Partial ordering }` 🚫

https://en.wikipedia.org/wiki/Partially_ordered_set

Compare Examples

```
vector<string> v = { ... };

sort(v.begin(), v.end());

sort(v.begin(), v.end(), less<>());

sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
  return s1 < s2;
});

sort(v.begin(), v.end(), [](const string & s1, const string & s2)
{
  return stricmp(s1.c_str(), s2.c_str()) < 0;
});
```

Compare Examples

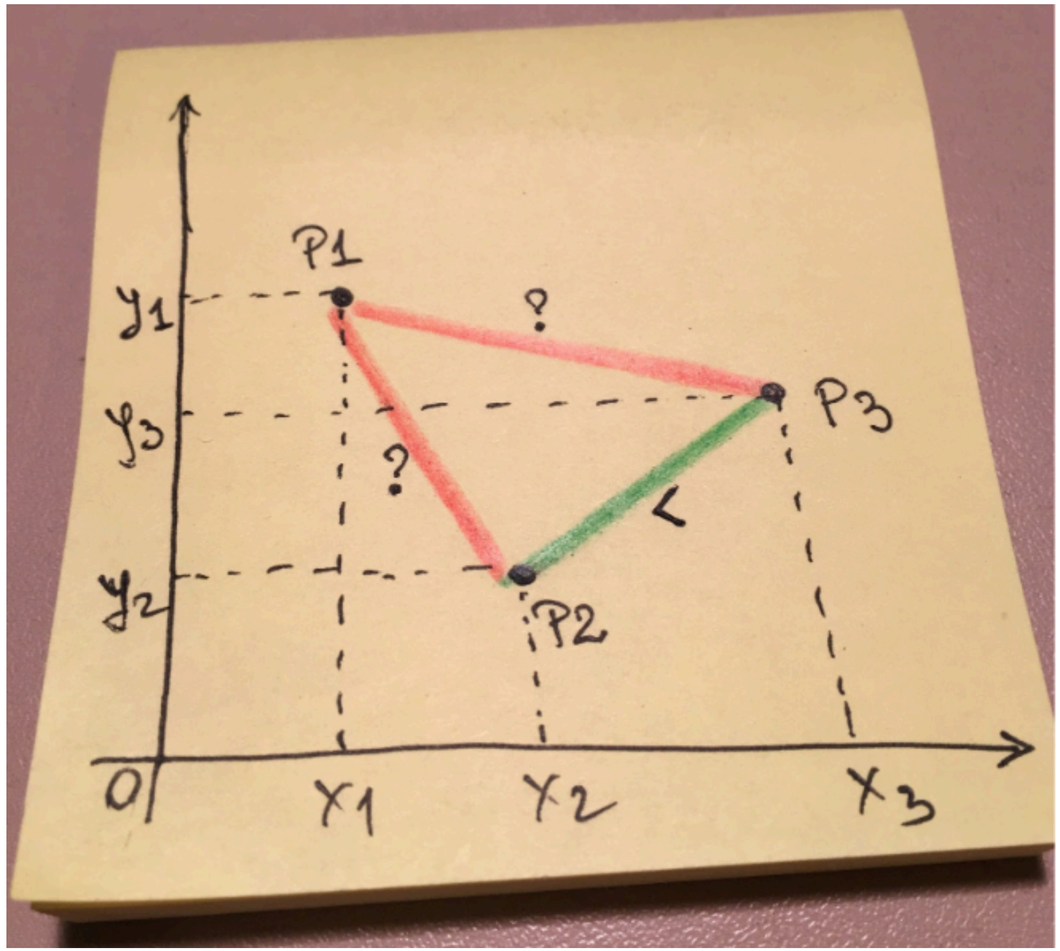
Initially, students go for this predicate:

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
});
```

Is this a good Compare predicate for 2D points ?

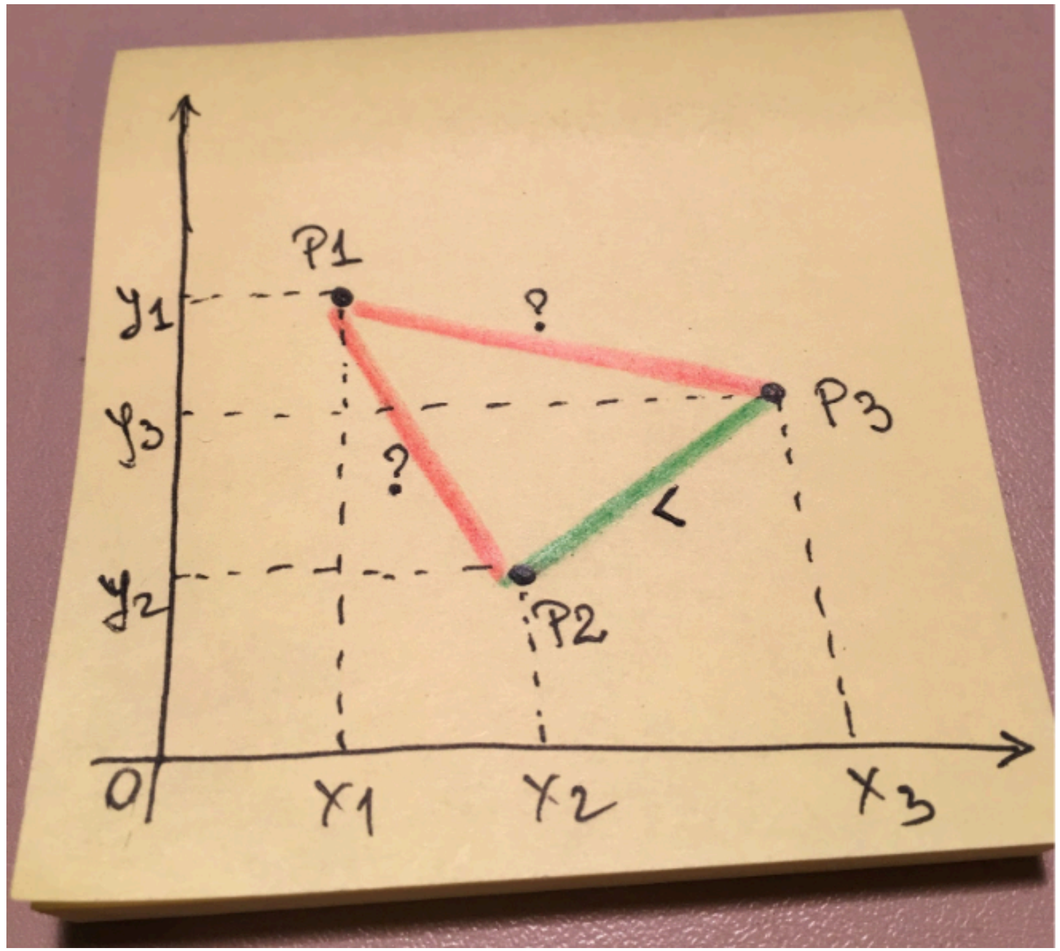
Compare Examples

```
Let { P1, P2, P3 }
x1 < x2; y1 > y2;
x1 < x3; y1 > y3;
x2 < x3; y2 < y3;

auto comp = [](const Point & p1,
               const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}
```



Compare Examples

```
Let { P1, P2, P3 }
x1 < x2; y1 > y2;
x1 < x3; y1 > y3;
x2 < x3; y2 < y3;

auto comp = [](const Point & p1,
               const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}

=>
```




```
P2 and P1 are unordered (P2 ? P1) | comp(P2,P1)==false && comp(P1,P2)==false
P1 and P3 are unordered (P1 ? P3) | comp(P1,P3)==false && comp(P3,P1)==false
P2 and P3 are ordered   (P2 < P3) | comp(P2,P3)==true  && comp(P3,P2)==false
```

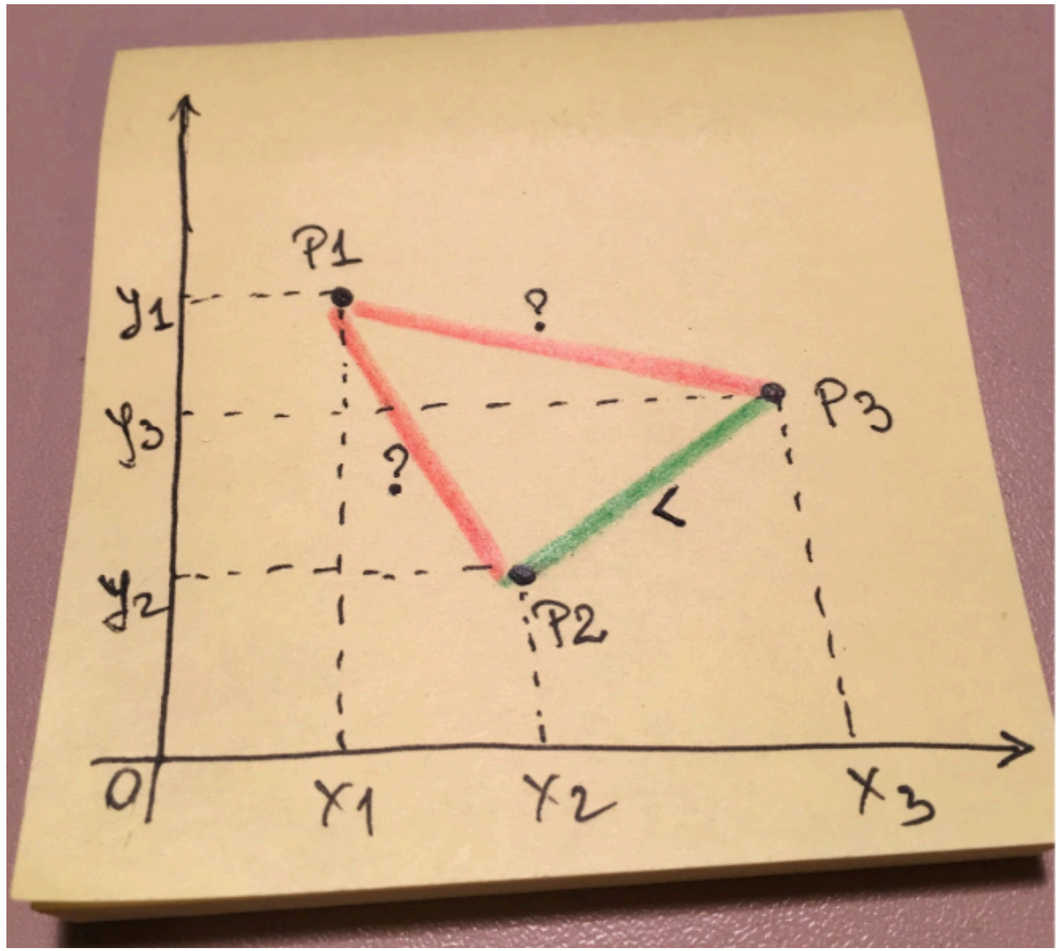
Compare Examples

Definition:

```
if comp(a,b)==false && comp(b,a)==false
=> a and b are equivalent
```

```
auto comp = [](const Point & p1,
               const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}

=>
```

```
P2 is equivalent to P1
P1 is equivalent to P3
P2 is less than     P3
```

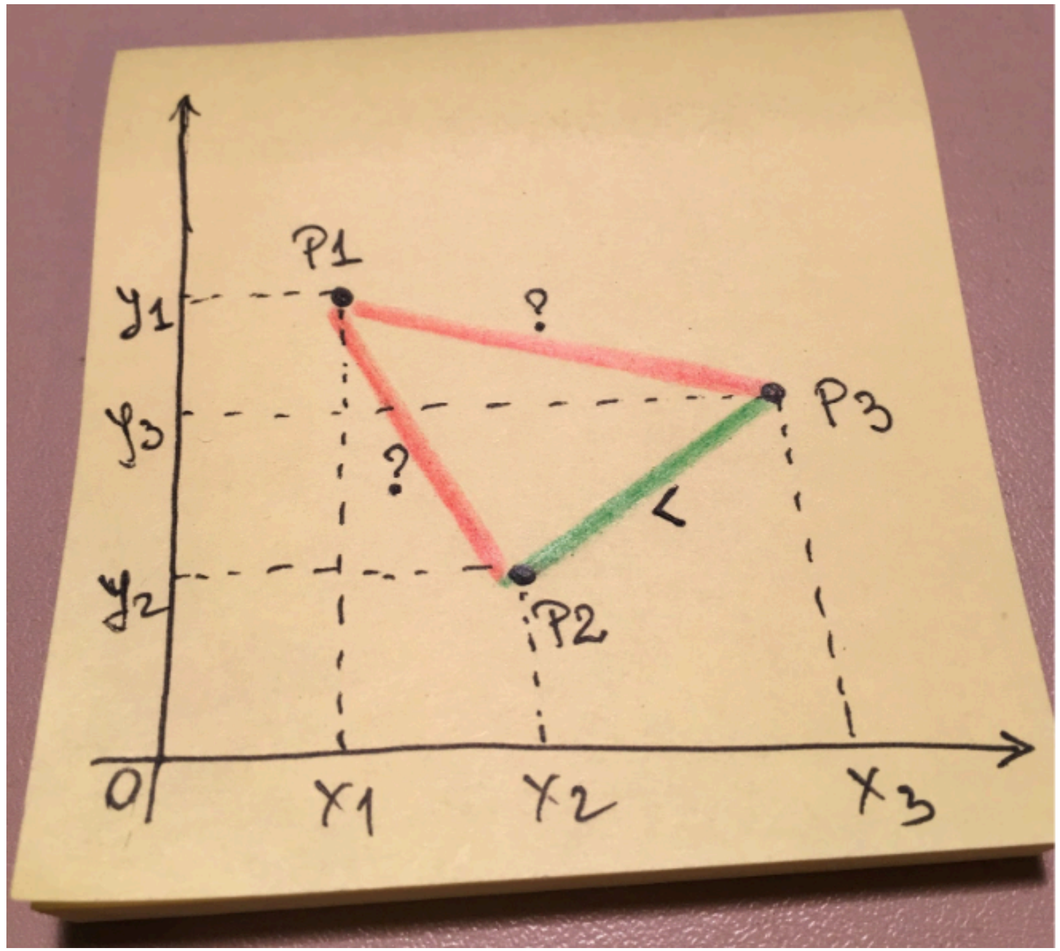


Compare Examples

Definition:

```
if comp(a,b)==false && comp(b,a)==false
=> a and b are equivalent
```

```
auto comp = [](const Point & p1,
              const Point & p2)
{
  return (p1.x < p2.x) && (p1.y < p2.y);
}

=>

P2 is equivalent to P1
P1 is equivalent to P3
P2 is less than     P3
```



`Compare` Concept

Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**

`Compare` Concept

Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**

Strict weak ordering = Partial ordering + *Transitivity of Equivalence*

`Compare` Concept

Partial ordering **relationship is not enough** 🤔

`Compare` **needs a *stronger* constraint**

Strict weak ordering = Partial ordering + *Transitivity of Equivalence*

where:

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

Strict weak ordering

https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

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
```

Total ordering relationship

`comp()` induces a ***strict total ordering***
on the equivalence classes determined by `equiv()`

https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

Total ordering relationship

`comp()` induces a ***strict total ordering***
on the equivalence classes determined by `equiv()`

The equivalence relation and its equivalence classes
partition the elements of the set,
and are **totally ordered** by **<**

https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

Compare Examples

Eventually, students gravitate towards this model:

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  // compare distance from origin
  return (p1.x * p1.x + p1.y * p1.y) <
         (p2.x * p2.x + p2.y * p2.y);
});
```

Compare Examples

It takes some back and forth discussions to lead students to comparing by parts

```
struct Point { int x; int y; };
vector<Point> v = { ... };

sort(v.begin(), v.end(), [](const Point & p1, const Point & p2)
{
  if (p1.x < p2.x) return true;
  if (p2.x < p1.x) return false;

  return p1.y < p2.y;
});
```

This is a really good Compare predicate for 2D points

`Compare` Examples

The general idea is to pick an order in which to compare elements/parts of the object.

(we first compared by **X** coordinate, and then by **Y** coordinate for *equivalent* **X**)

`Compare` Examples

The general idea is to pick an order in which to compare elements/parts of the object.

(we first compared by **X** coordinate, and then by **Y** coordinate for *equivalent* **X**)

This strategy is analogous to how a dictionary works,

so it is often called **dictionary order** or **lexicographical order**.

`Compare` Examples

The general idea is to pick an order in which to compare elements/parts of the object.

(we first compared by **X** coordinate, and then by **Y** coordinate for *equivalent* **X**)

This strategy is analogous to how a dictionary works,

so it is often called **dictionary order** or **lexicographical order**.

`std::pair<T, U>` defines the six comparison operators
in terms of the corresponding operators of the pair's ***components***

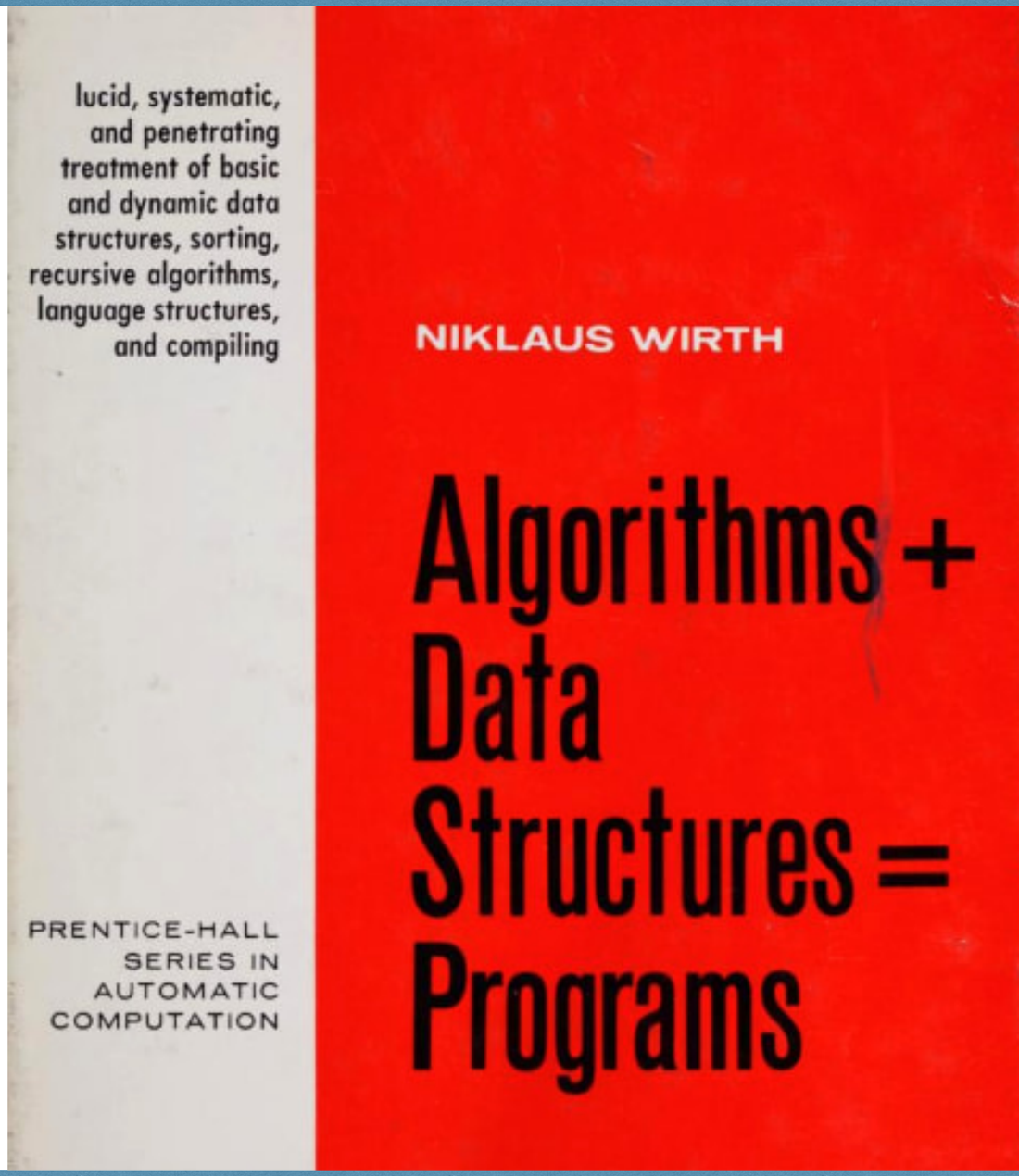
Tired

The difference between Efficiency and Performance

Why do we care ?

Because: *“Software is getting slower more rapidly than hardware becomes faster.”*

“A Plea for Lean Software” - *Niklaus Wirth*



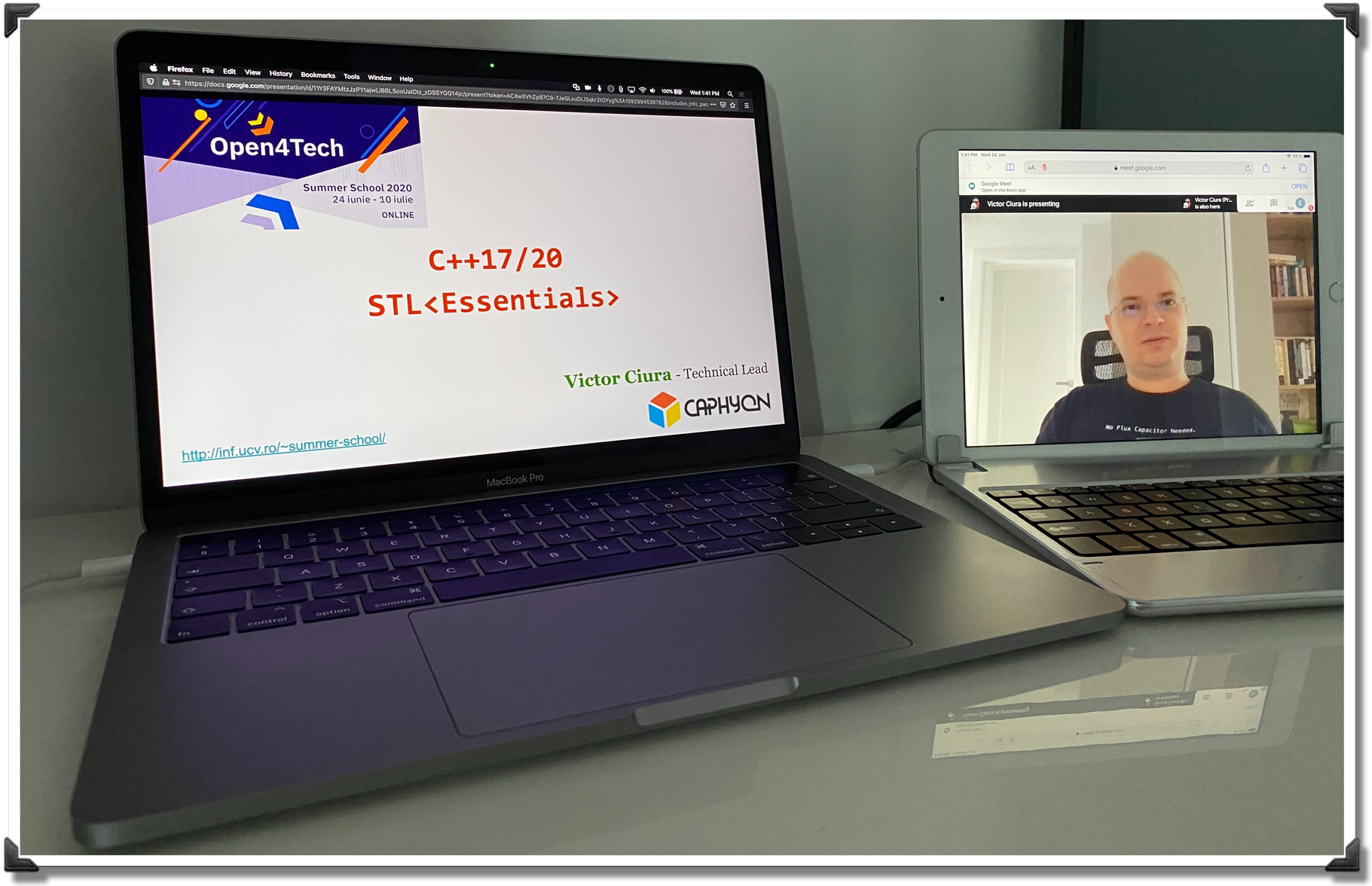
The difference between Efficiency and Performance

Efficiency	Performance
the amount of work you need to do	how fast you can do that work
governed by your algorithm	governed by your data structures

ℹ️ Efficiency and performance are not necessarily dependent on one another.

2020-21

C++ 17/20 STL Essentials



Compare Examples

Definition:

```
if comp(a,b)==false && comp(b,a)==false
=> a and b are equivalent
```

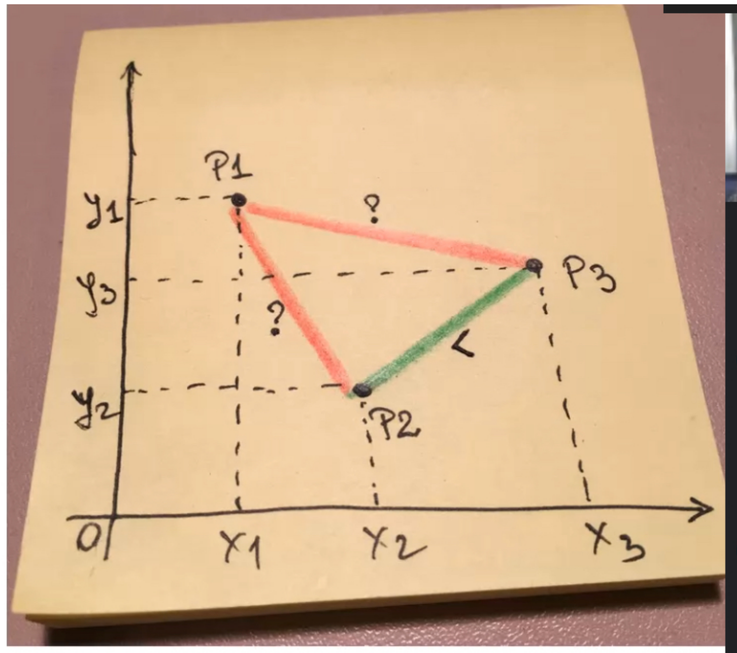
```
Let { P1, P2, P3 }
x1 < x2; y1 > y2;
x1 < x3; y1 > y3;
x2 < x3; y2 < y3;
```

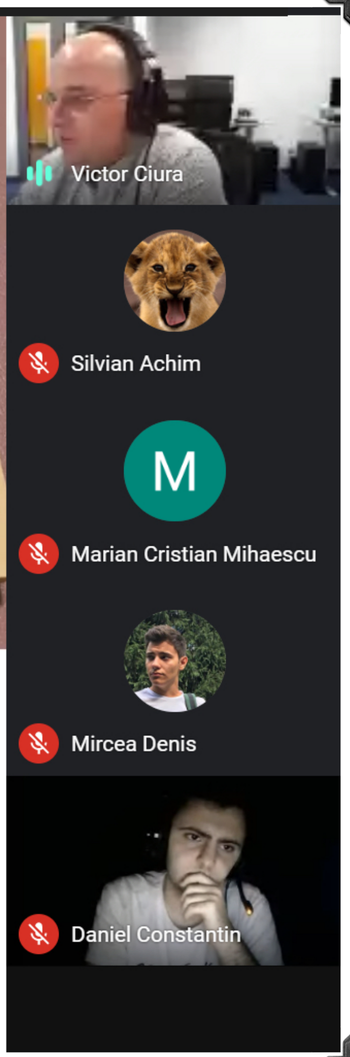
=>

```
P2 and P1 are unordered (P2?P1) comp(P2,P1)==false && comp(P1,P2)==false
P1 and P3 are unordered (P1?P3) comp(P1,P3)==false && comp(P3,P1)==false
P2 and P3 are ordered   (P2<P3) comp(P2,P3)==true  && comp(P3,P2)==false
```

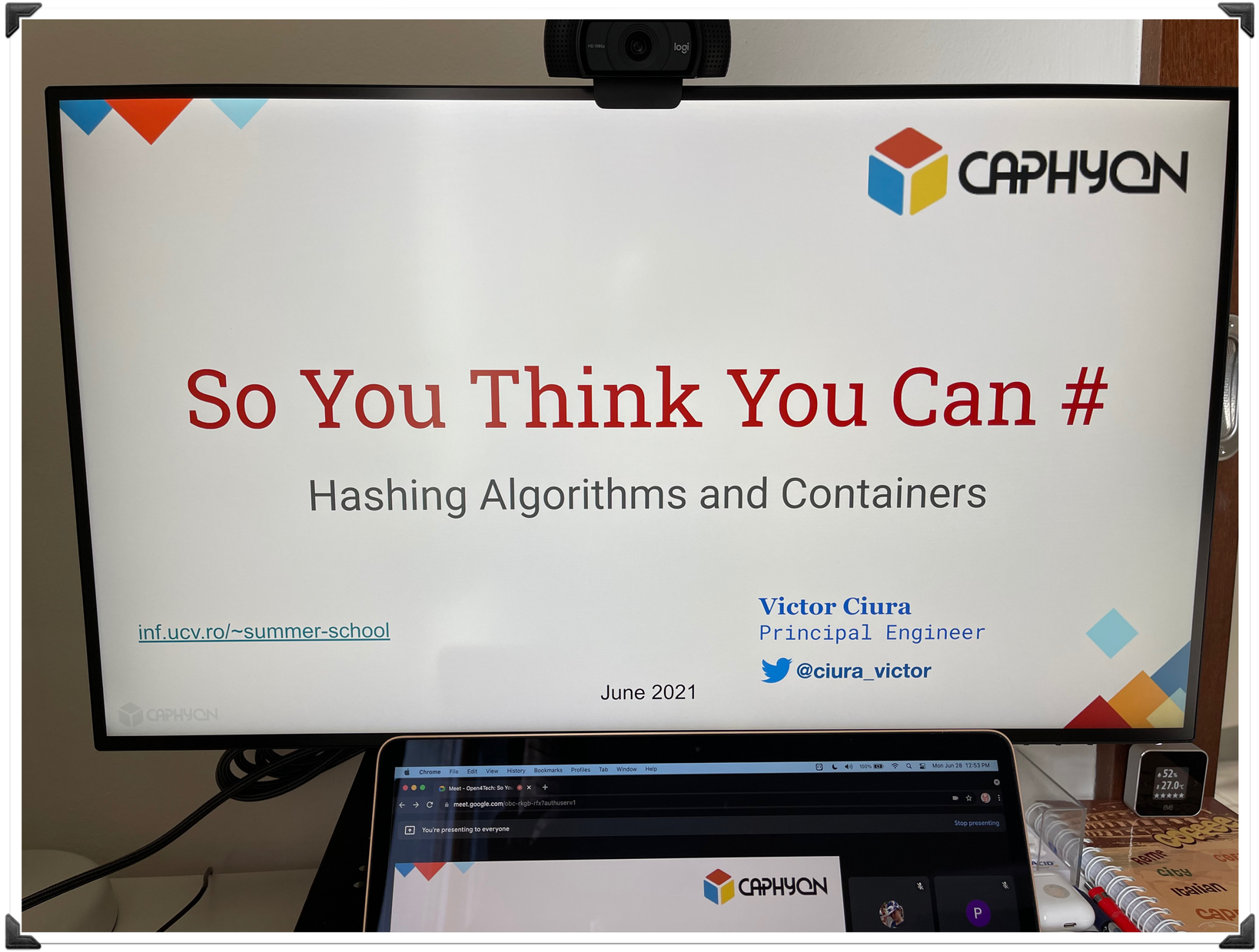
=>

```
P2 is equivalent to P1
P1 is equivalent to P3
P2 is less than P3
```





So you think you can



ONLINE Challenges

People (31) Chat

Add people

- Gabriel Diaconita (You)
- ALEXANDRA-ȘTEFANIA TUȚAC
- ALEXANDRU-COSTINEL STAN...
- ALIN-PETRE IOVA
- Andrei Marcu
- ANDREI MARCU
- Badoi Mircea
- BOGDAN DRĂGHICI
- Bogdan Marius Muga
- Bogdan-Andrei Zanfir
- CRISTIAN-MIHAI MILCU
- CRISTIANA - MĂDĂLINA PROD...
- Cristiana Costache
- Daniel Constantin
- DANIEL-ALEXANDRU IOVA

🎓 ONLINE Challenges

When you have a meeting @ WFH, usually

everyone turns on their camera

🎓 ONLINE Challenges

When you have a meeting @ WFH, usually
everyone turns on their camera

In workshops for companies, some trainers claim that
50-70% of attendees have the camera on

🎓 ONLINE Challenges

When you have a meeting @ WFH, usually
everyone turns on their camera

In workshops for companies, some trainers claim that
50-70% of attendees have the camera on

In open workshops (paid) the camera on is
about 20-50%

🎓 ONLINE Challenges

When you have a meeting @ WFH, usually
everyone turns on their camera

In workshops for companies, some trainers claim that
50-70% of attendees have the camera on

In open workshops (paid) the camera on is
about 20-50%

In 🎓 UNI courses/seminars, my friends in academia (and myself)
report an average of ~10% students with camera on

Beyond 2021

Is this a lost cause?

Is this a lost cause?

I think not.

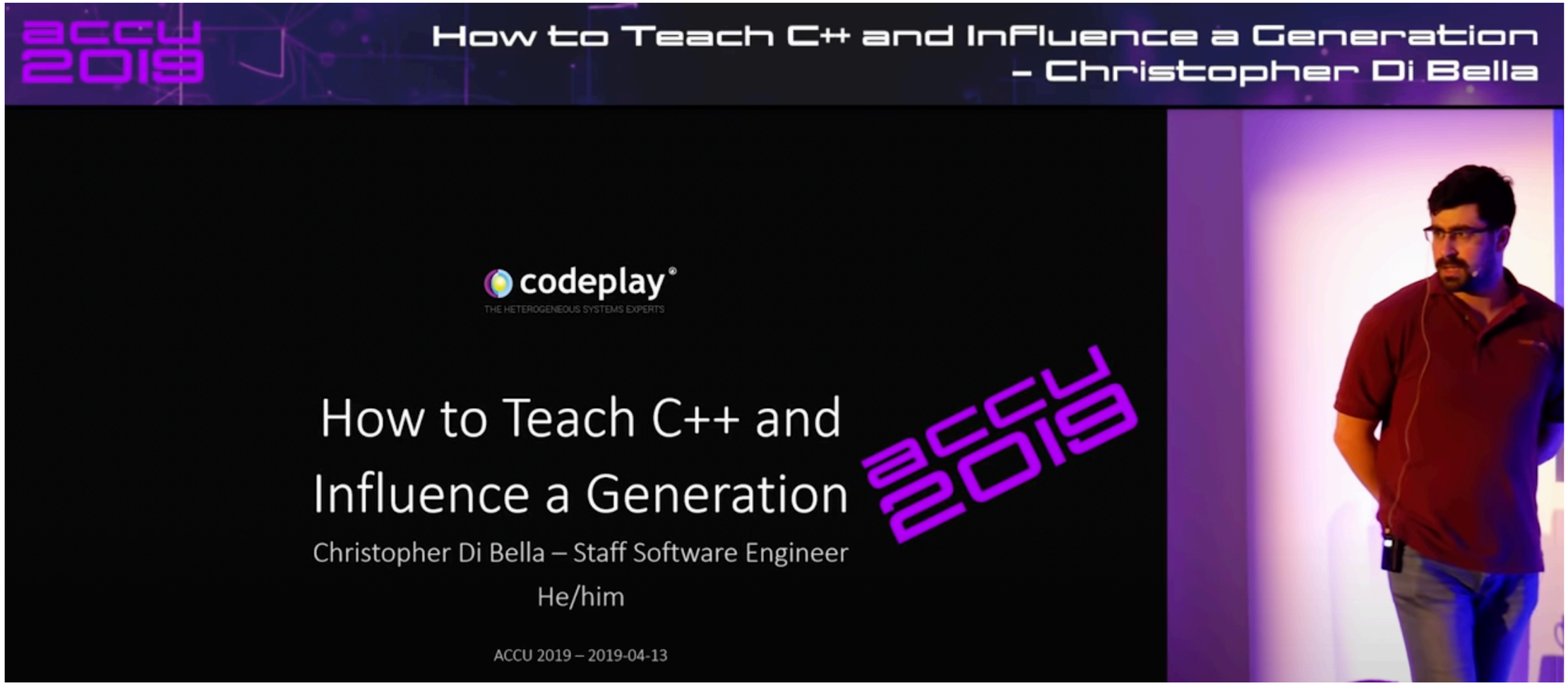
Is this a lost cause?

I think not.

Modern C++ is simpler and safer and we have numerous opportunities to make it more teachable at the same time.

ISO WG21 - SG 20 : Education

You can get involved : SG 20



www.youtube.com/watch?v=nzEPHkUxXZs

The king is dead, long live the king!

Free discussion

C++ 🎓 UNIverse

hopin.com/events/mini-conference-with-victor-ciura

Victor Ciura
Principal Engineer