From C to C++20 and Beyond

An Evolution of 50 Years

Nicolai M. Josuttis

josuttis.com

@NicoJosuttis

10/20

C++

josuttis | eckstein
IT communication

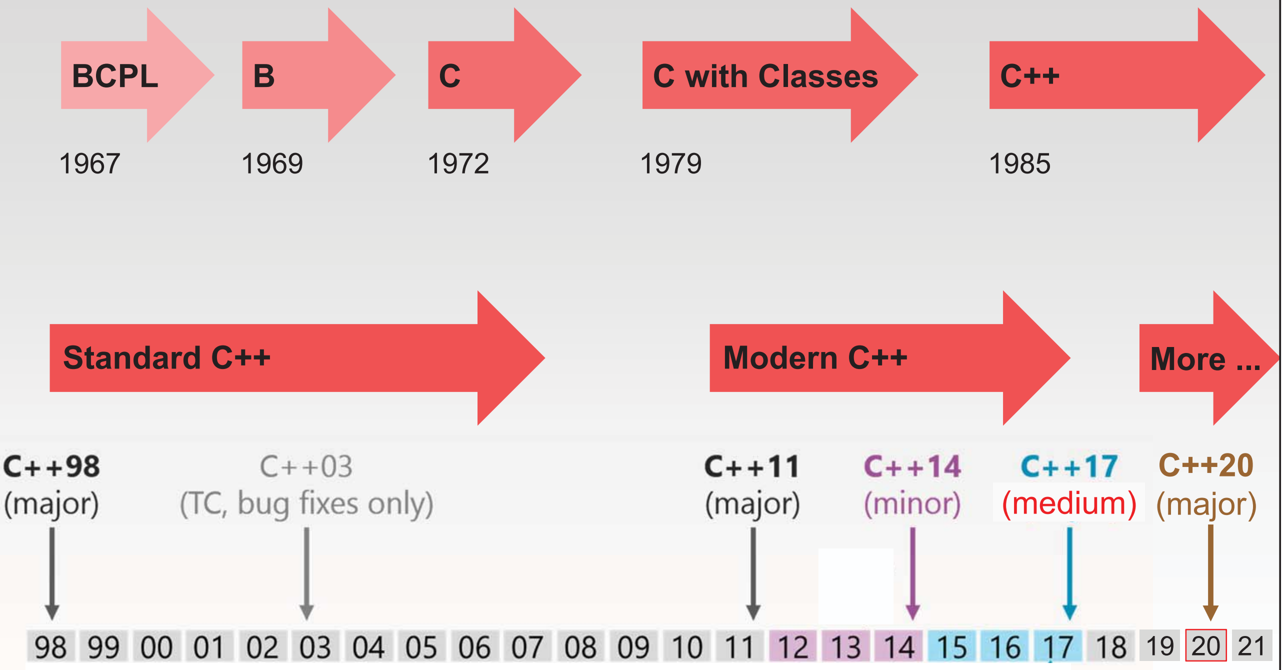
Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - X and OSF/Motif
  - SOA

Die C++-Standardbibliothek
Objektorientiertes Programmieren in C++
The C++ Standard Library A Tutorial and Reference
C++11
C++ Templates The Complete Guide
The C++ Standard Library Second Edition A Tutorial and Reference
C++17 The Complete Guide
Nicolai M. Josuttis
C++ Move Semantics The Complete Guide
C++ Templates The Complete Guide Second Edition

@NicoJosuttis

C++ Timeline



Evolution of People

- **Bjarne Stroustrup**
- **The C++ Standards Committee**

Evolution of Hardware and Software

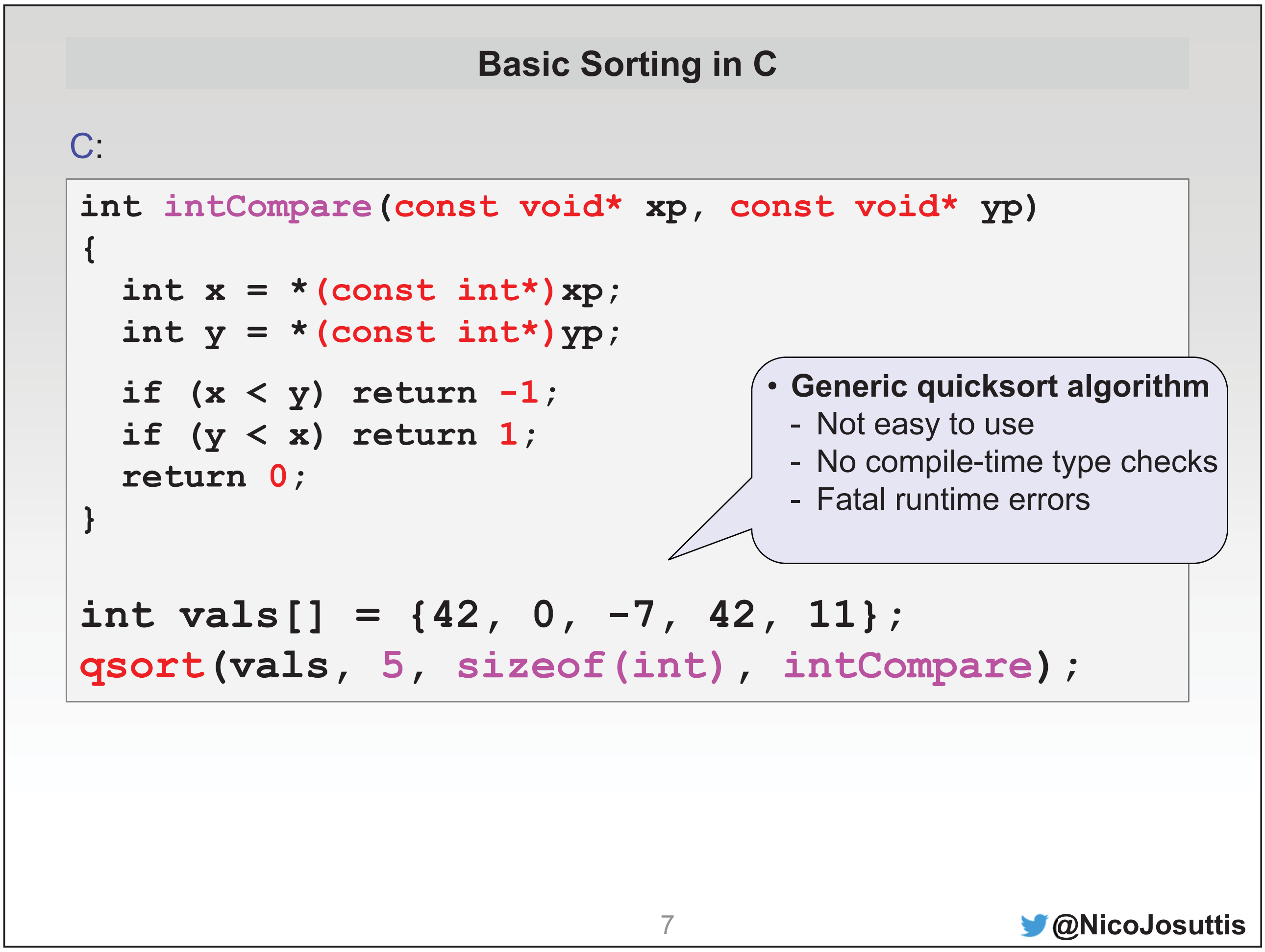
- Single core
- More and more memory
- More and more transistors
- Multi core
- The end of Moore's law
- Threads
- CPU caches
- Out-of-order execution
- SIMD (single instruction for multiple data)
- ...
- More and more data
- More and more complexity
- ...

Backward Compatibility 😈

Evolution of Usability

C++

josuttis | eckstein IT communication

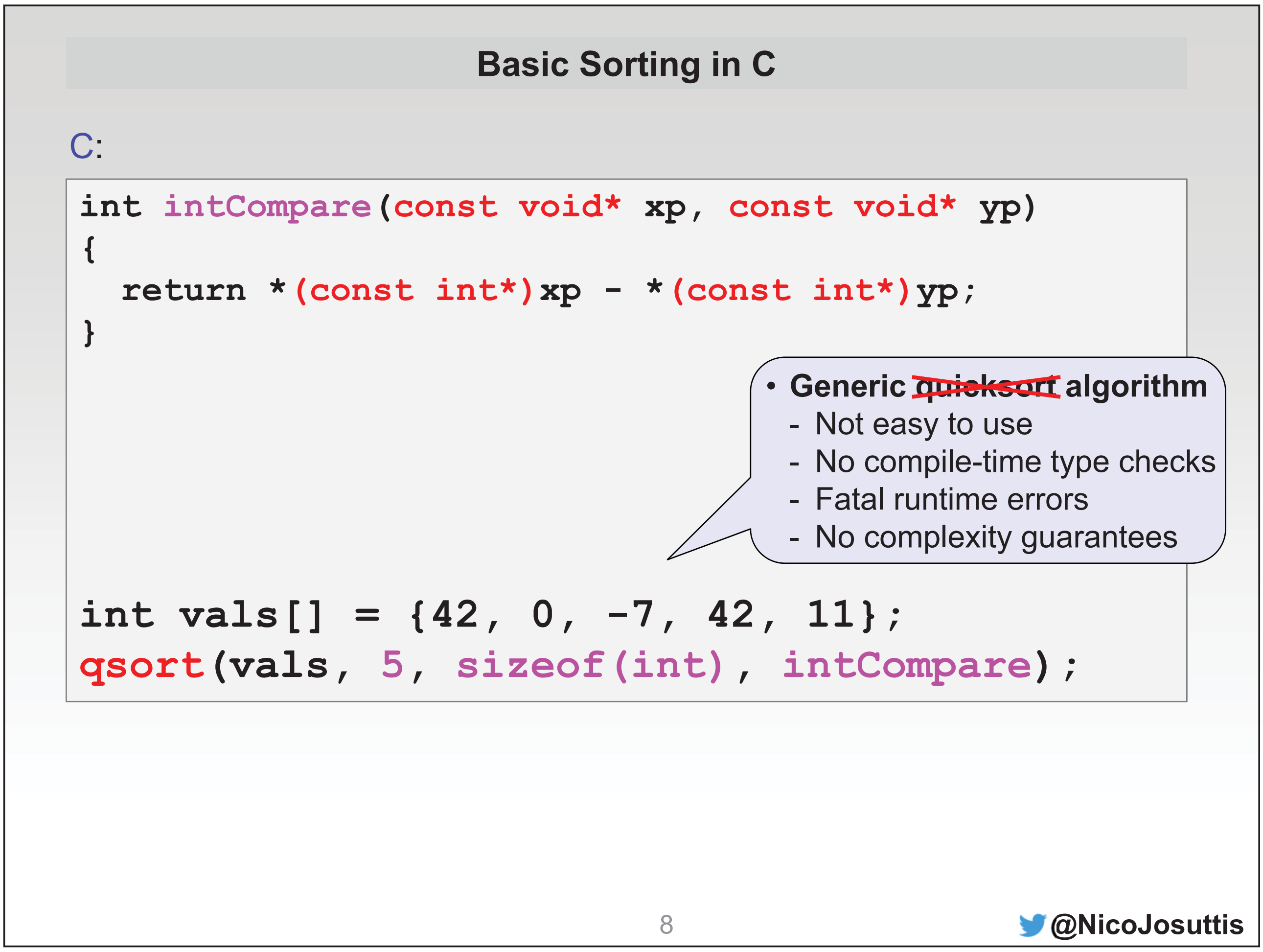


Basic Sorting in C

C:

```
int intCompare(const void* xp, const void* yp)
{
  int x = *(const int*)xp;
  int y = *(const int*)yp;

  if (x < y) return -1;
  if (y < x) return 1;
  return 0;
}

int vals[] = {42, 0, -7, 42, 11};
qsort(vals, 5, sizeof(int), intCompare);
```

- **Generic quicksort algorithm**
  - Not easy to use
  - No compile-time type checks
  - Fatal runtime errors

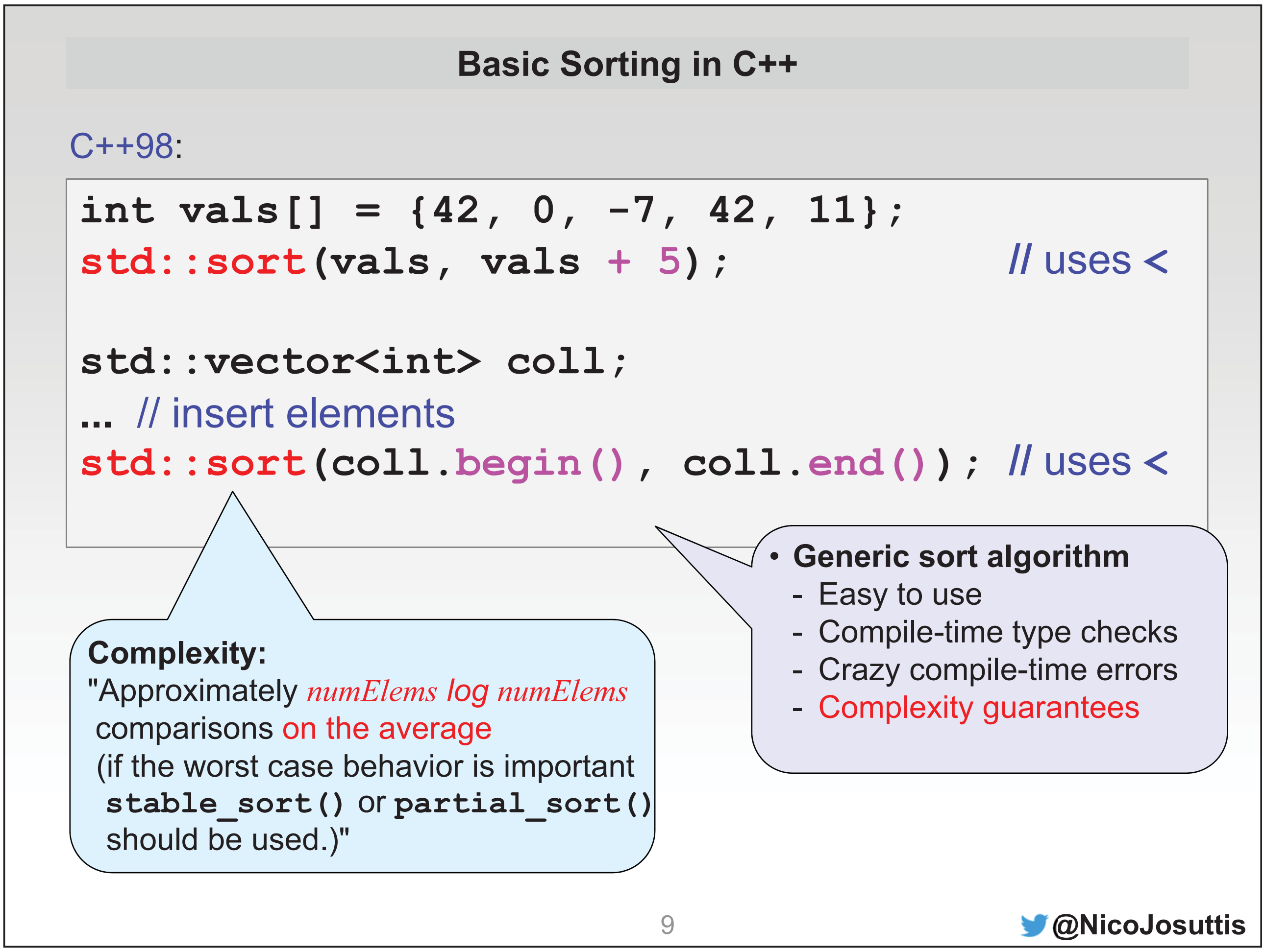


Basic Sorting in C

C:

```
int intCompare(const void* xp, const void* yp)
{
  return *(const int*)xp - *(const int*)yp;
}

int vals[] = {42, 0, -7, 42, 11};
qsort(vals, 5, sizeof(int), intCompare);
```

- **Generic ~~quicksort~~ algorithm**
  - Not easy to use
  - No compile-time type checks
  - Fatal runtime errors
  - No complexity guarantees



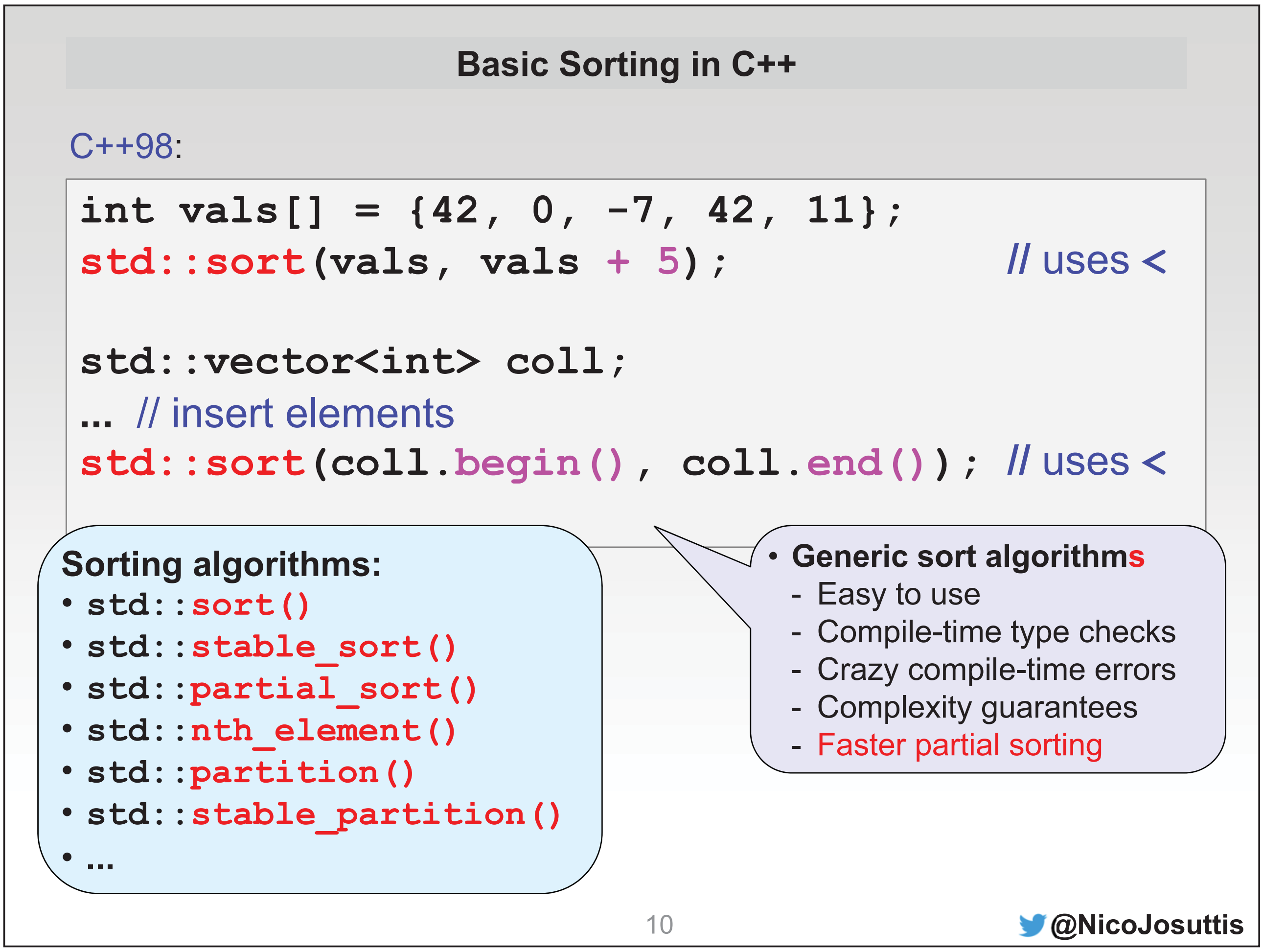
Basic Sorting in C++

C++98:

```
int vals[] = {42, 0, -7, 42, 11};
std::sort(vals, vals + 5);                 // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end()); // uses <
```

Complexity:
"Approximately *numElems* log *numElems* comparisons on the average (if the worst case behavior is important `stable_sort()` or `partial_sort()` should be used.)"

- **Generic sort algorithm**
  - Easy to use
  - Compile-time type checks
  - Crazy compile-time errors
  - Complexity guarantees



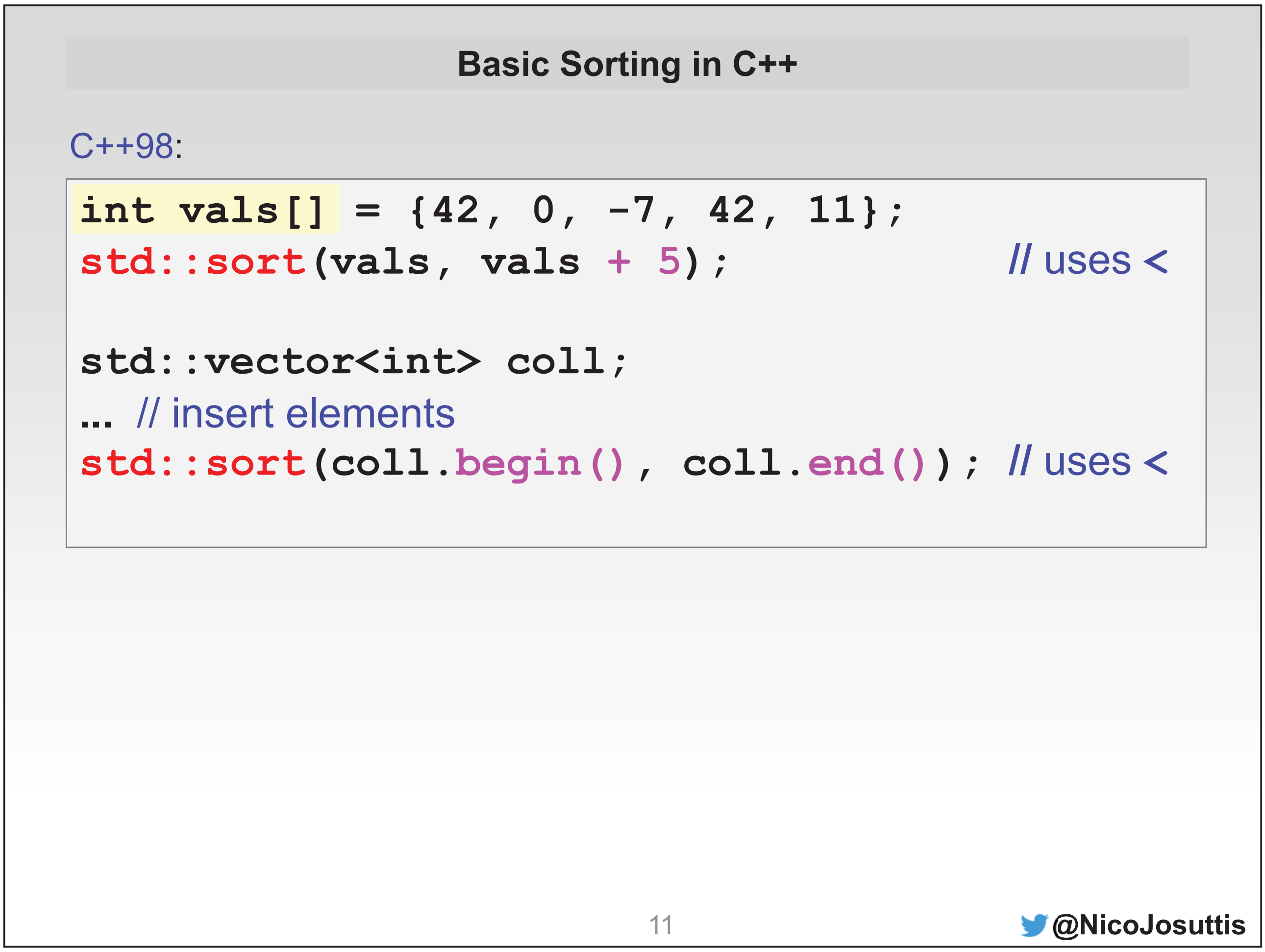
Basic Sorting in C++

C++98:

```
int vals[] = {42, 0, -7, 42, 11};
std::sort(vals, vals + 5);                 // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end()); // uses <
```

Sorting algorithms:
- `std::sort()`
- `std::stable_sort()`
- `std::partial_sort()`
- `std::nth_element()`
- `std::partition()`
- `std::stable_partition()`
- ...

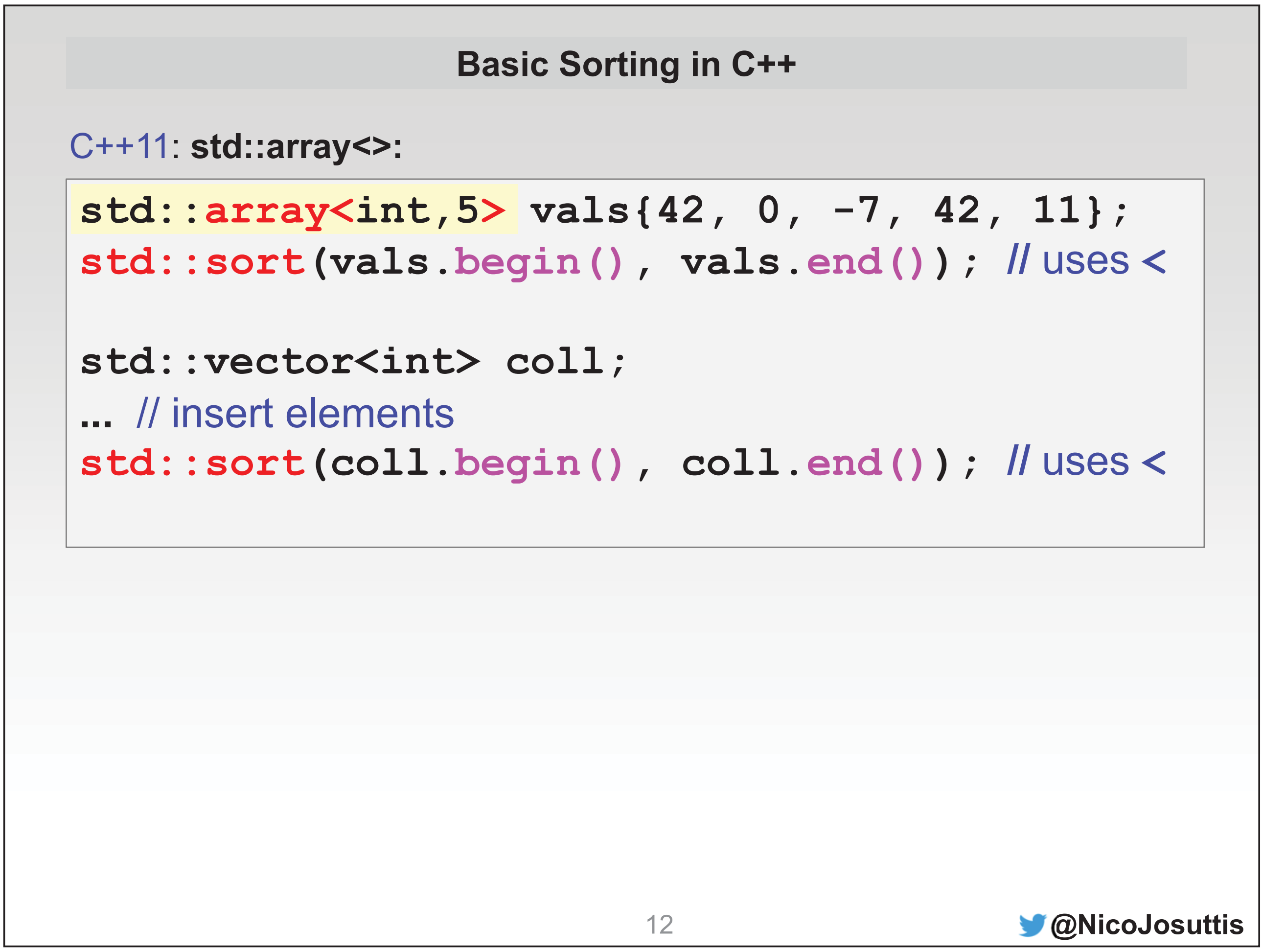
- **Generic sort algorithms**
  - Easy to use
  - Compile-time type checks
  - Crazy compile-time errors
  - Complexity guarantees
  - Faster partial sorting



Basic Sorting in C++

C++98:

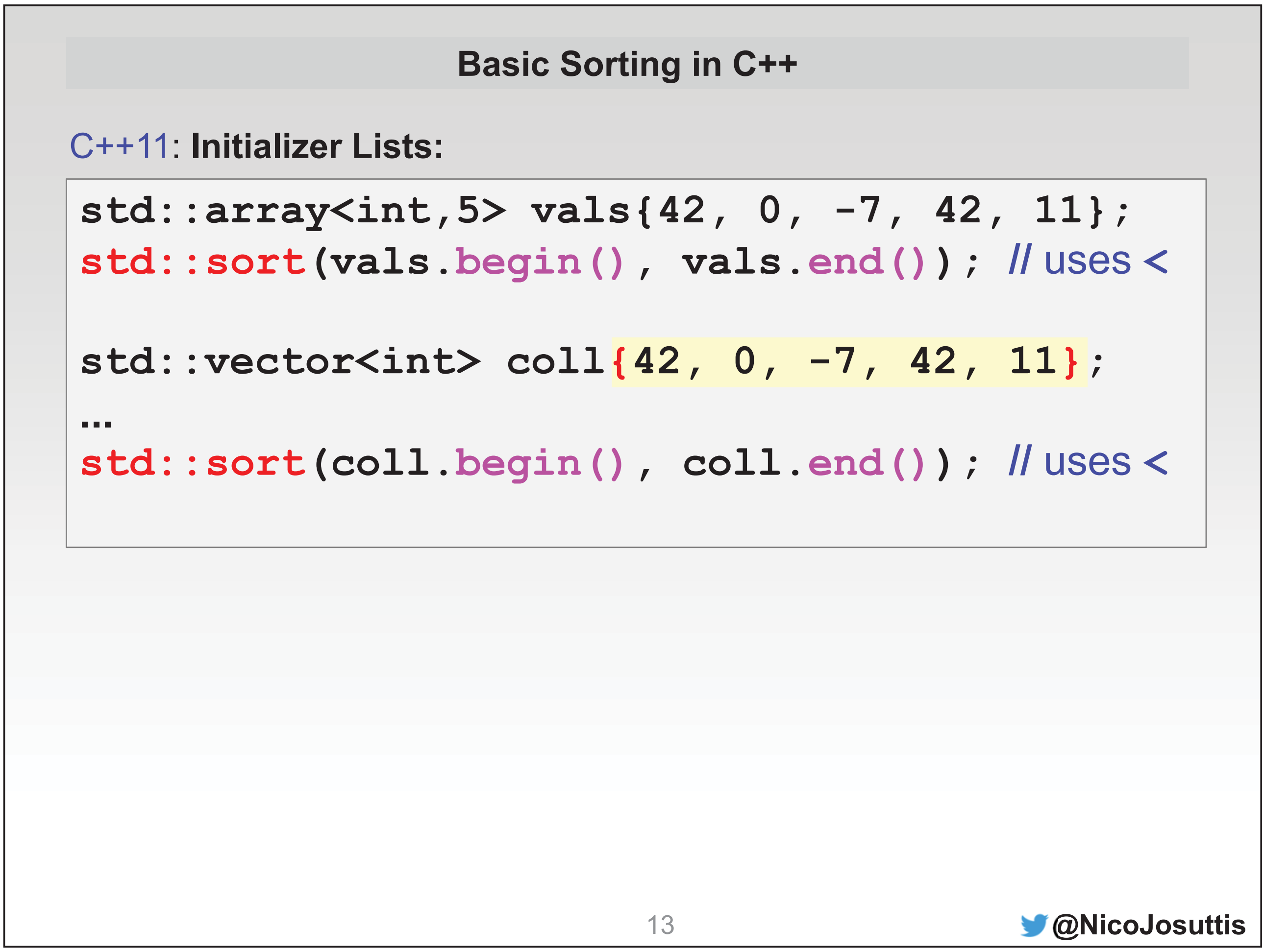
```
int vals[] = {42, 0, -7, 42, 11};
std::sort(vals, vals + 5);              // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end());    // uses <
```



Basic Sorting in C++

C++11: std::array<>:

```
std::array<int,5> vals{42, 0, -7, 42, 11};
std::sort(vals.begin(), vals.end());    // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end());    // uses <
```



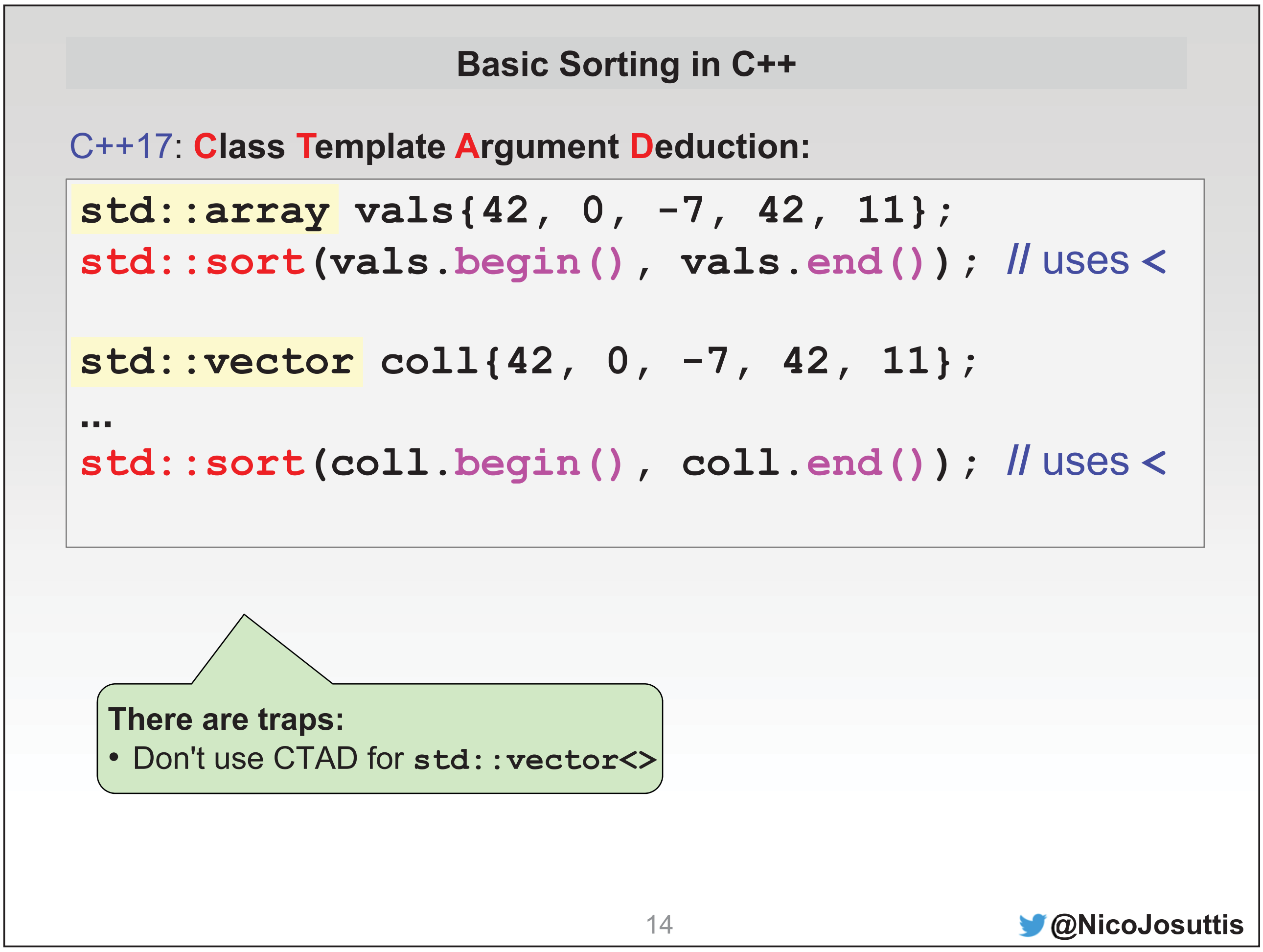
Basic Sorting in C++

C++11: **Initializer Lists:**

```
std::array<int,5> vals{42, 0, -7, 42, 11};
std::sort(vals.begin(), vals.end()); // uses <

std::vector<int> coll{42, 0, -7, 42, 11};
...
std::sort(coll.begin(), coll.end()); // uses <
```

13

@NicoJosuttis



Basic Sorting in C++

C++17: **Class Template Argument Deduction:**

```
std::array vals{42, 0, -7, 42, 11};
std::sort(vals.begin(), vals.end()); // uses <

std::vector coll{42, 0, -7, 42, 11};
...
std::sort(coll.begin(), coll.end()); // uses <
```

There are traps:

- Don't use CTAD for `std::vector<>`

14

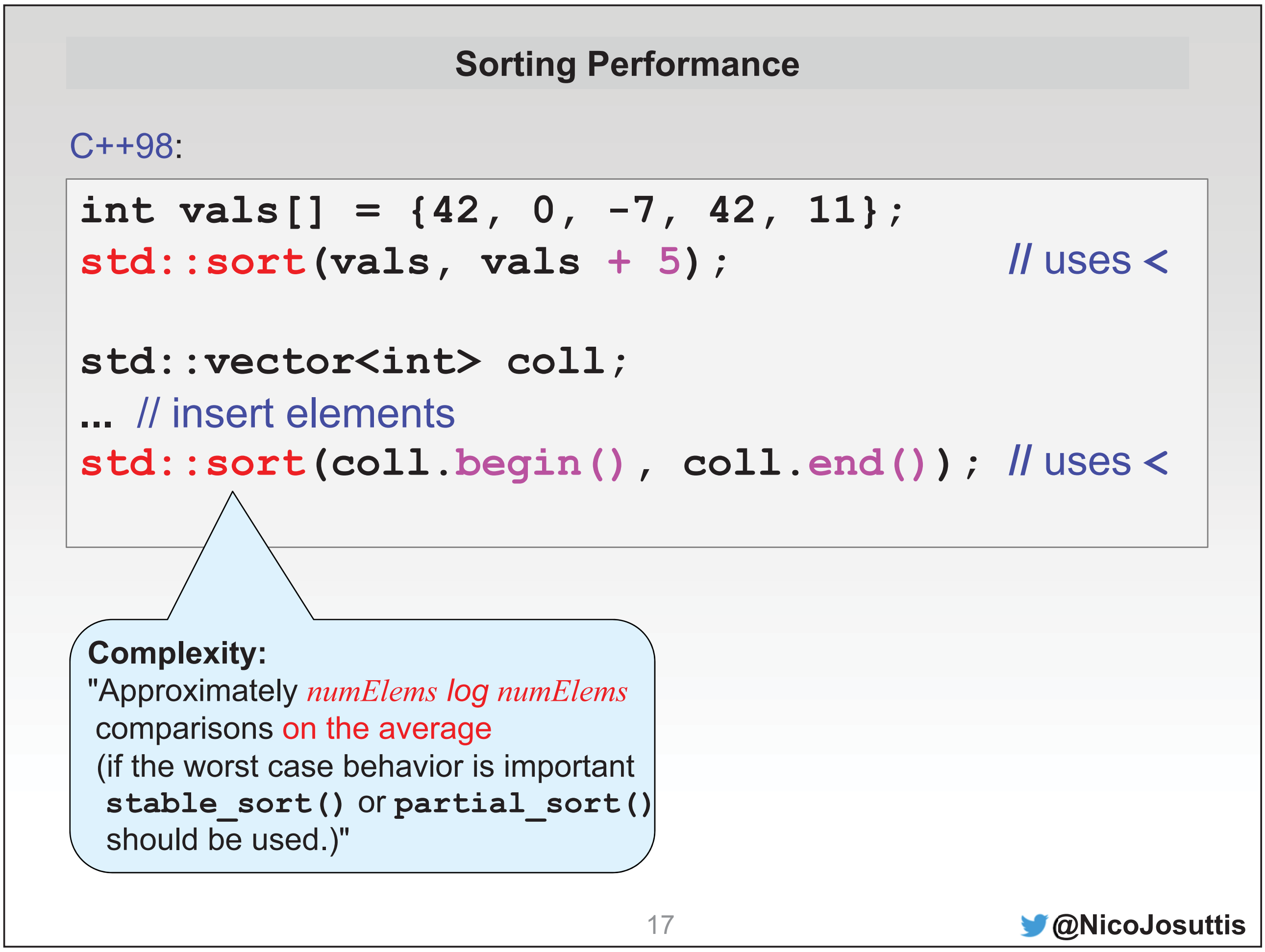
@NicoJosuttis

Basic Sorting in C++

C++20: **Ranges:**

```
std::array vals{42, 0, -7, 42, 11};
std::ranges::sort(vals);                      // uses <

std::vector coll{42, 0, -7, 42, 11};
...
std::ranges::sort(coll);                      // uses <
```

Evolution of Performance

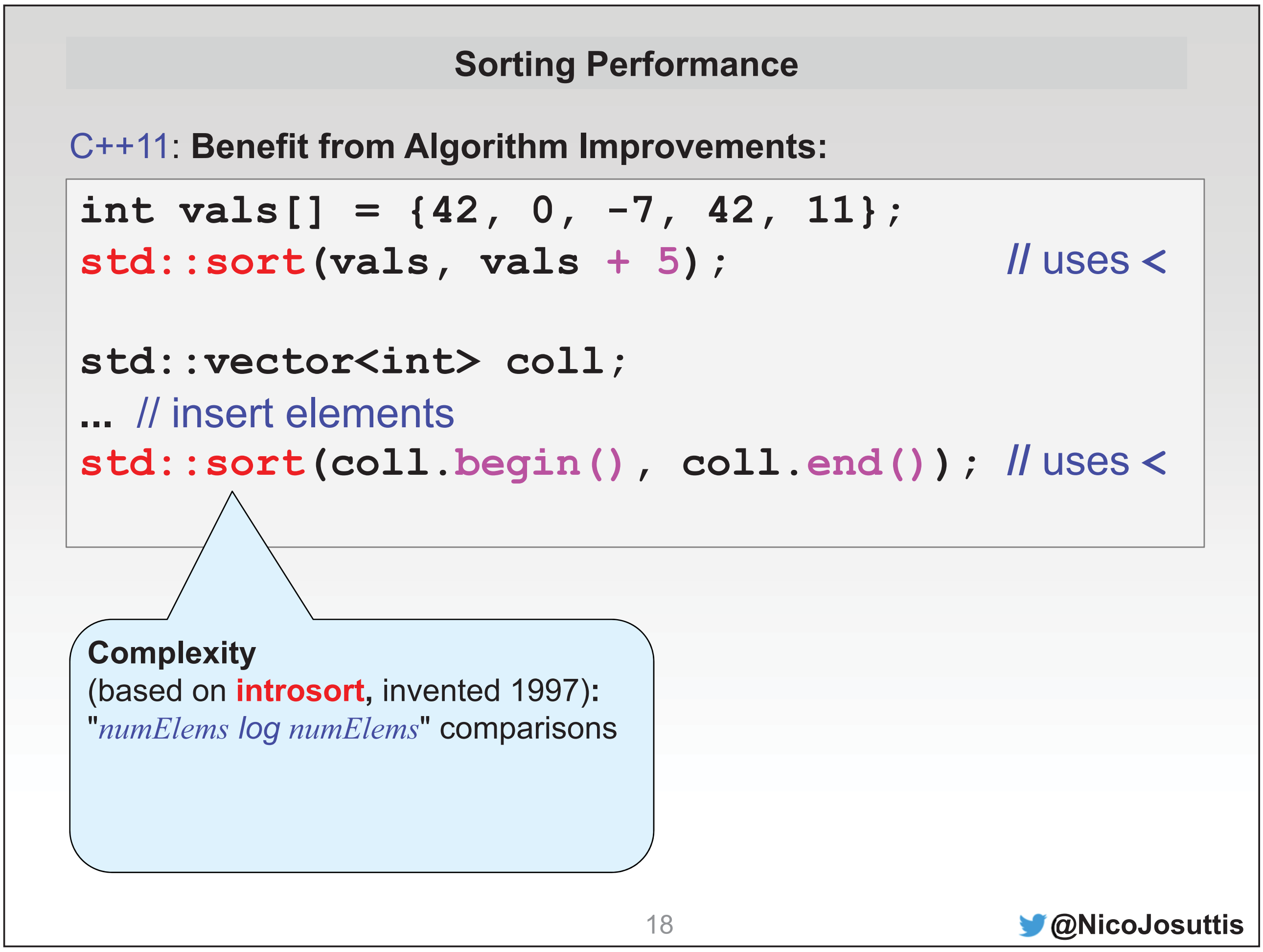


Sorting Performance

C++98:

```
int vals[] = {42, 0, -7, 42, 11};
std::sort(vals, vals + 5);                 // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end()); // uses <
```

Complexity:
"Approximately *numElems* log *numElems* comparisons on the average
(if the worst case behavior is important `stable_sort()` or `partial_sort()` should be used.)"

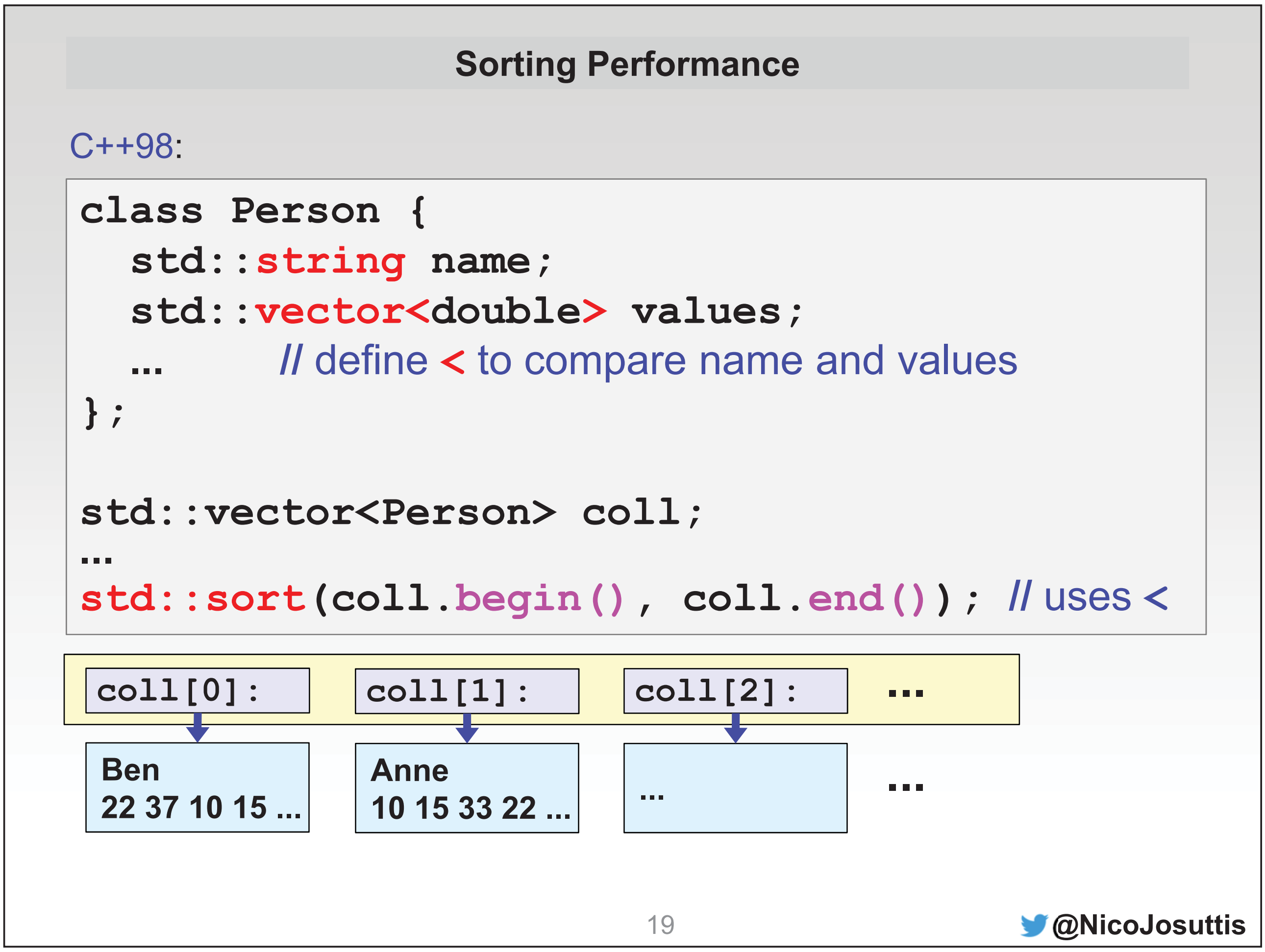


Sorting Performance

C++11: **Benefit from Algorithm Improvements:**

```
int vals[] = {42, 0, -7, 42, 11};
std::sort(vals, vals + 5);                 // uses <

std::vector<int> coll;
... // insert elements
std::sort(coll.begin(), coll.end()); // uses <
```

Complexity
(based on **introsort**, invented 1997):
"*numElems* log *numElems*" comparisons



Sorting Performance

C++98:

```
class Person {
   std::string name;
   std::vector<double> values;
   ...       // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // uses <
```

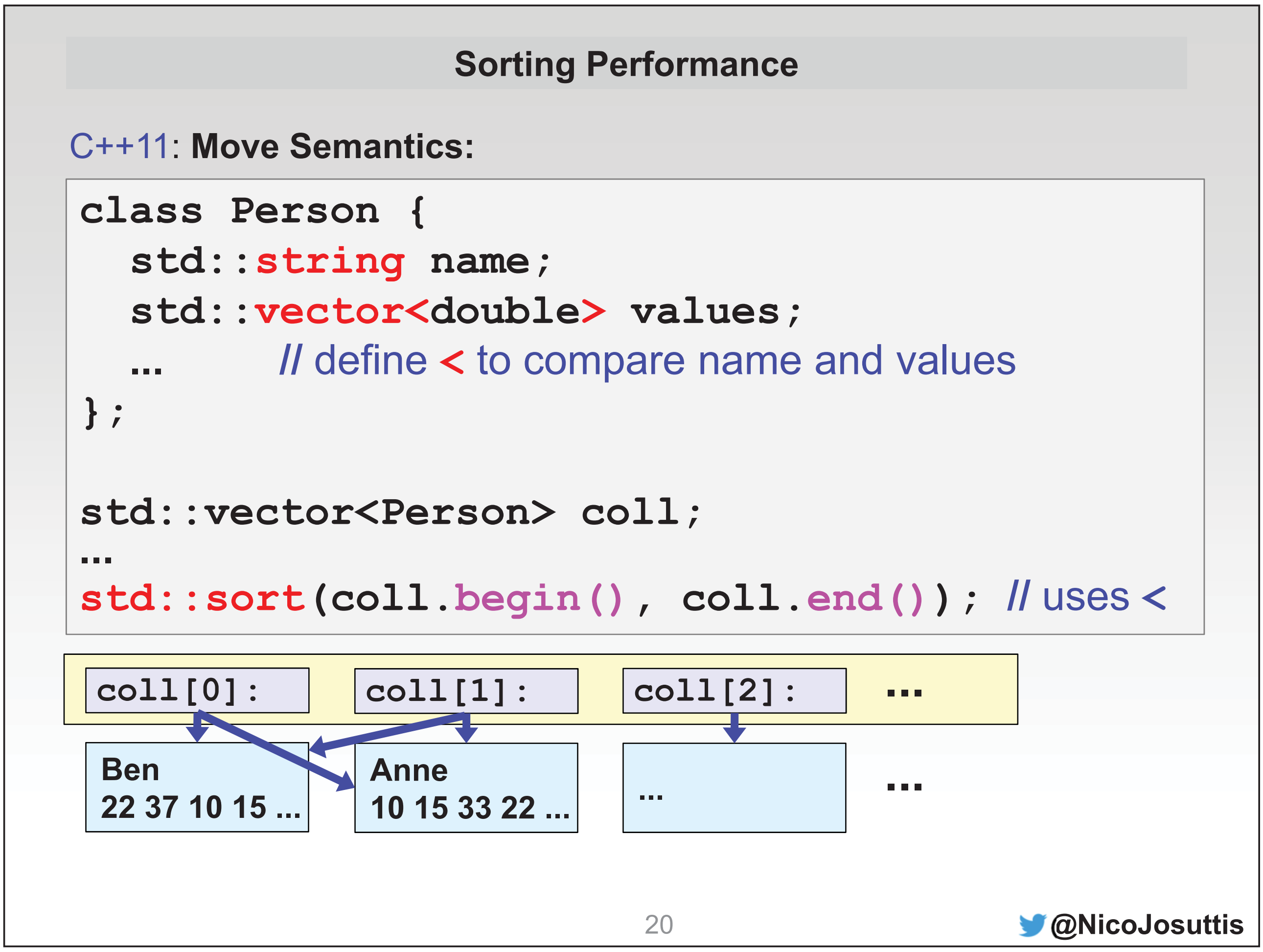
coll[0]: coll[1]: coll[2]: ...

Ben
22 37 10 15 ...

Anne
10 15 33 22 ...

...

...



Sorting Performance

C++11: **Move Semantics:**

```
class Person {
   std::string name;
   std::vector<double> values;
   ...       // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // uses <
```

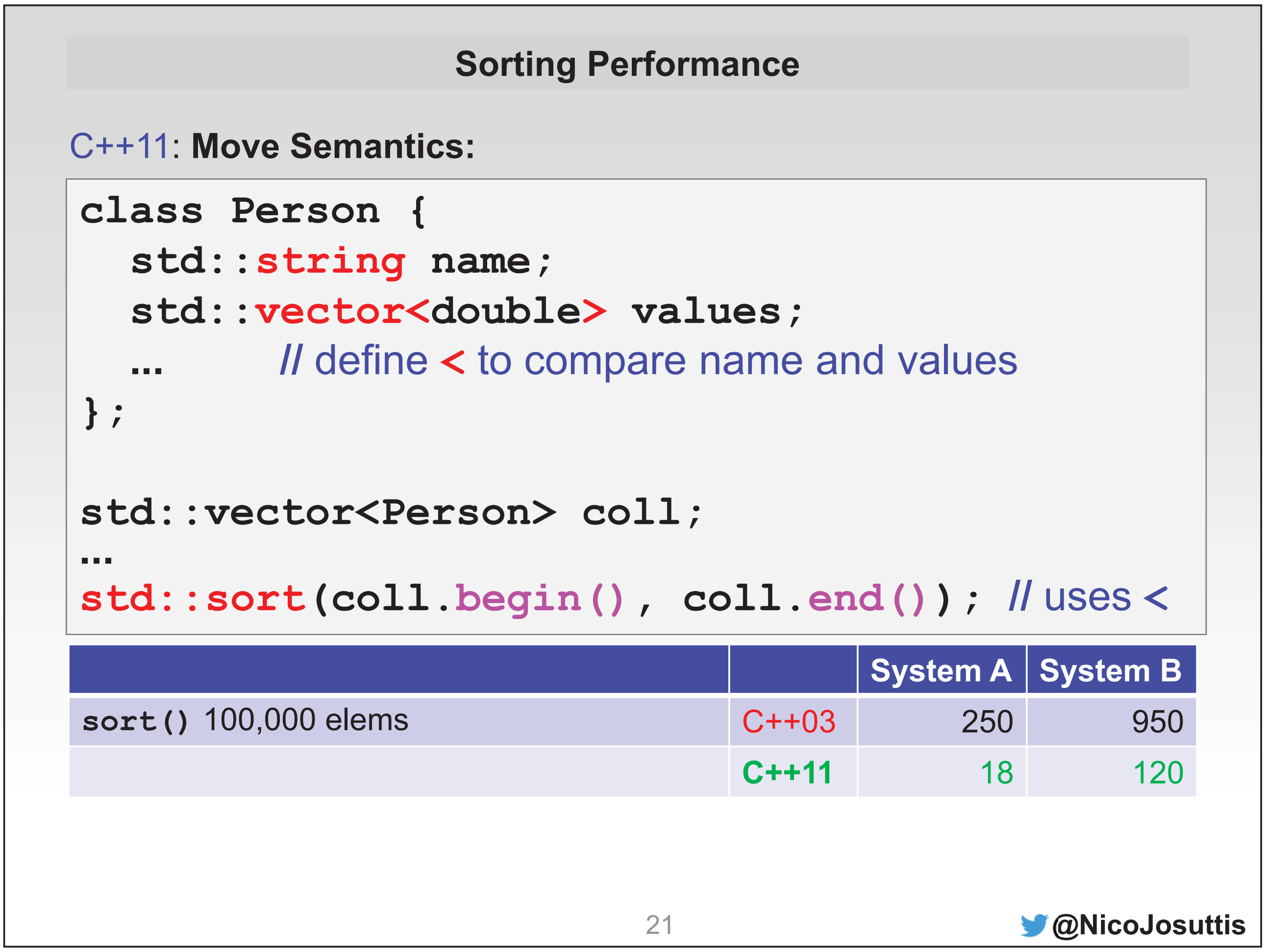
coll[0]: coll[1]: coll[2]: ...

Ben
22 37 10 15 ...

Anne
10 15 33 22 ...

...

...

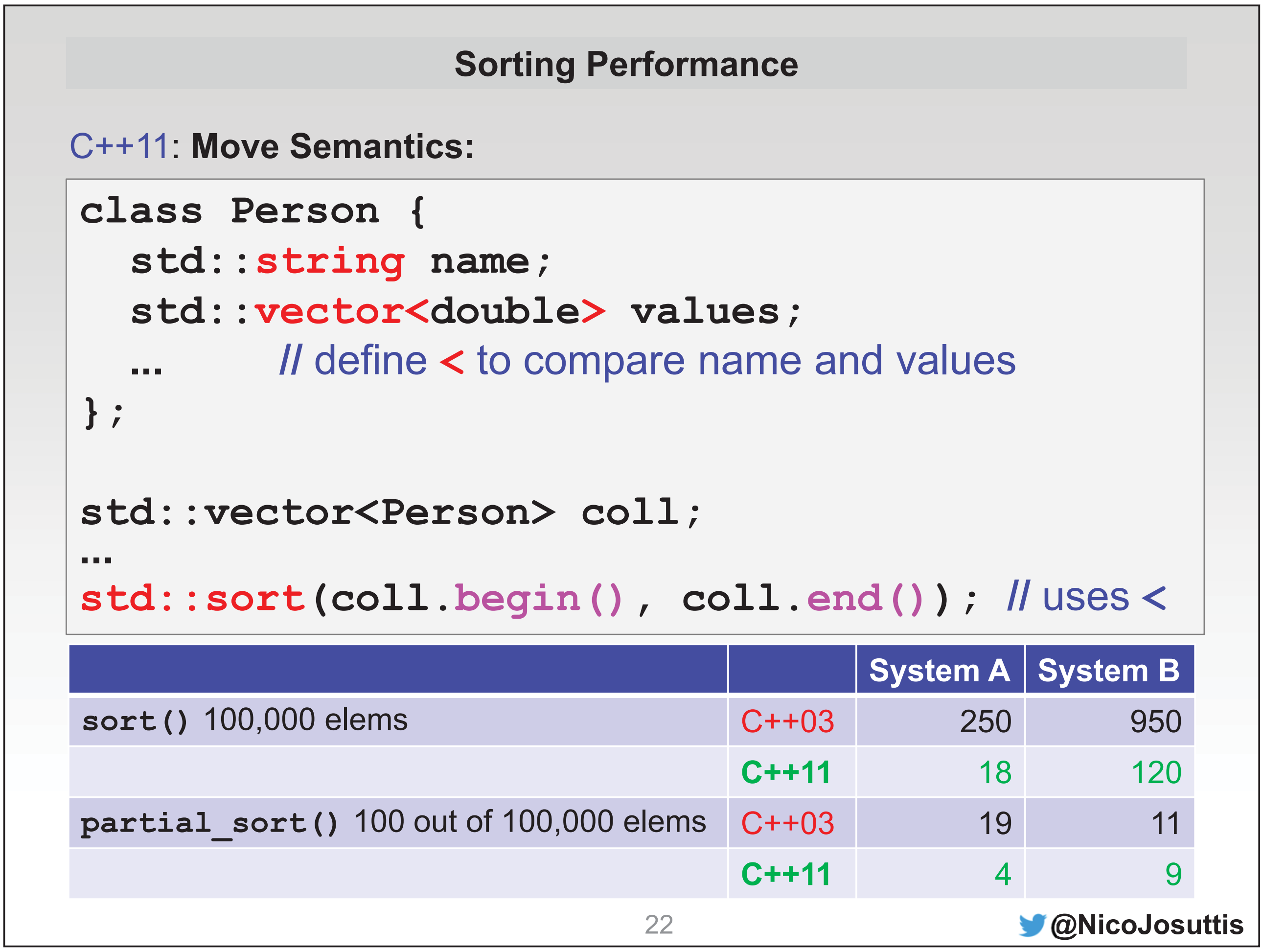


Sorting Performance

C++11: **Move Semantics:**

```
class Person {
   std::string name;
   std::vector<double> values;
   ...       // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // uses <
```

		System A	System B
`sort()` 100,000 elems	C++03	250	950
	C++11	18	120

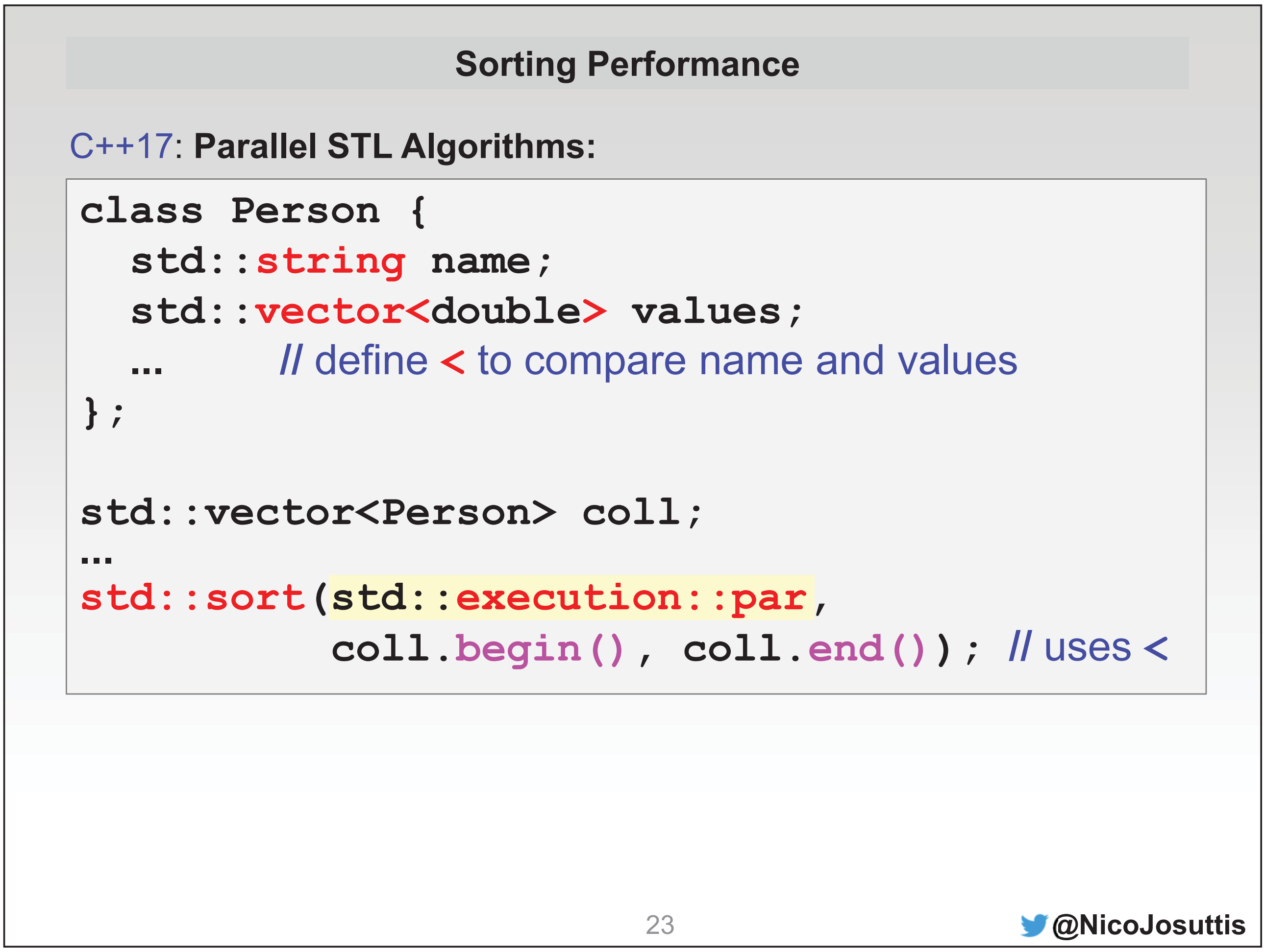


Sorting Performance

C++11: **Move Semantics:**

```
class Person {
   std::string name;
   std::vector<double> values;
   ...       // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // uses <
```

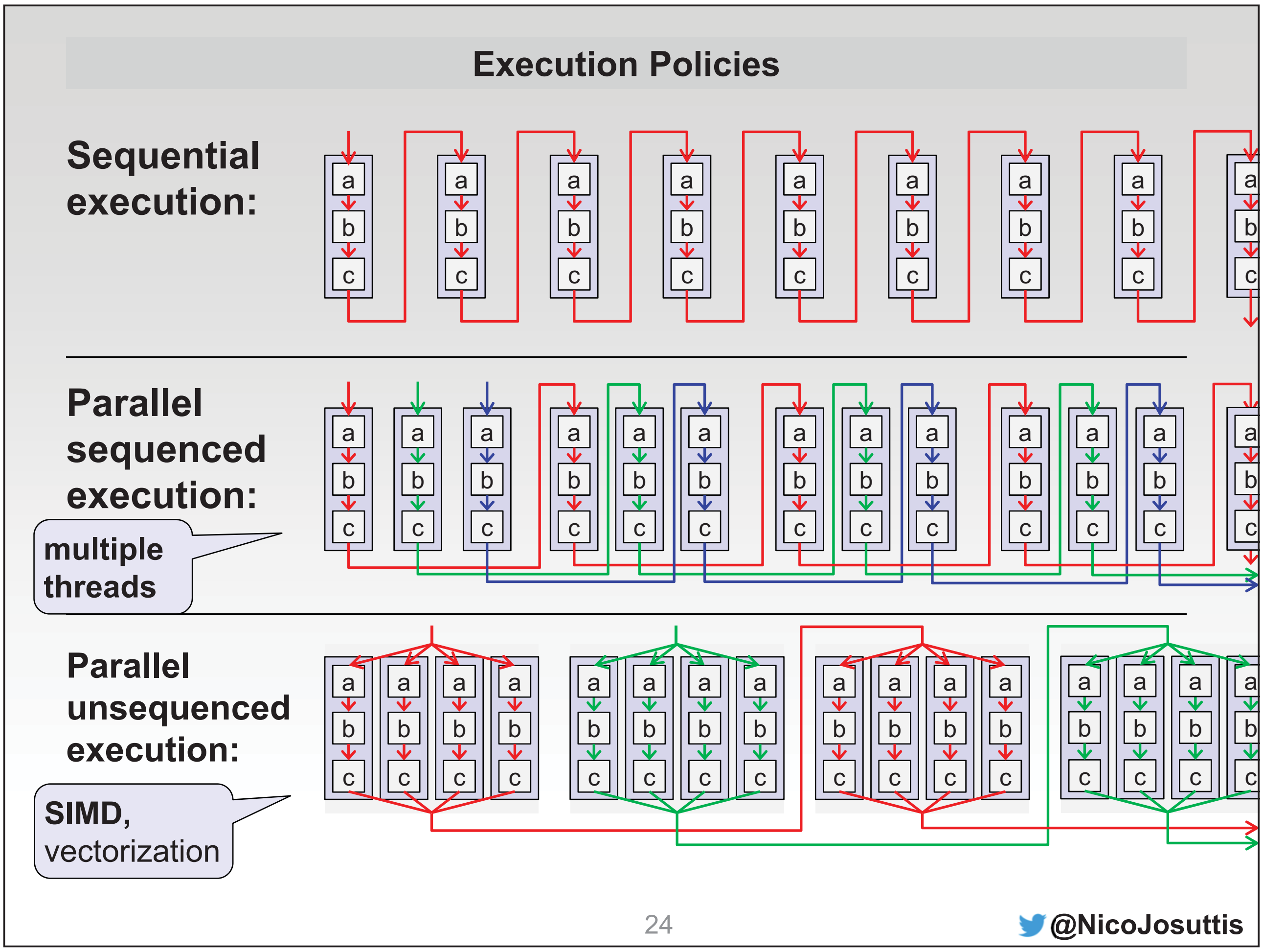
		System A	System B
`sort()` 100,000 elems	C++03	250	950
	C++11	18	120
`partial_sort()` 100 out of 100,000 elems	C++03	19	11
	C++11	4	9



Sorting Performance

C++17: **Parallel STL Algorithms:**

```
class Person {
   std::string name;
   std::vector<double> values;
   ...       // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(std::execution::par,
          coll.begin(), coll.end()); // uses <
```



Execution Policies

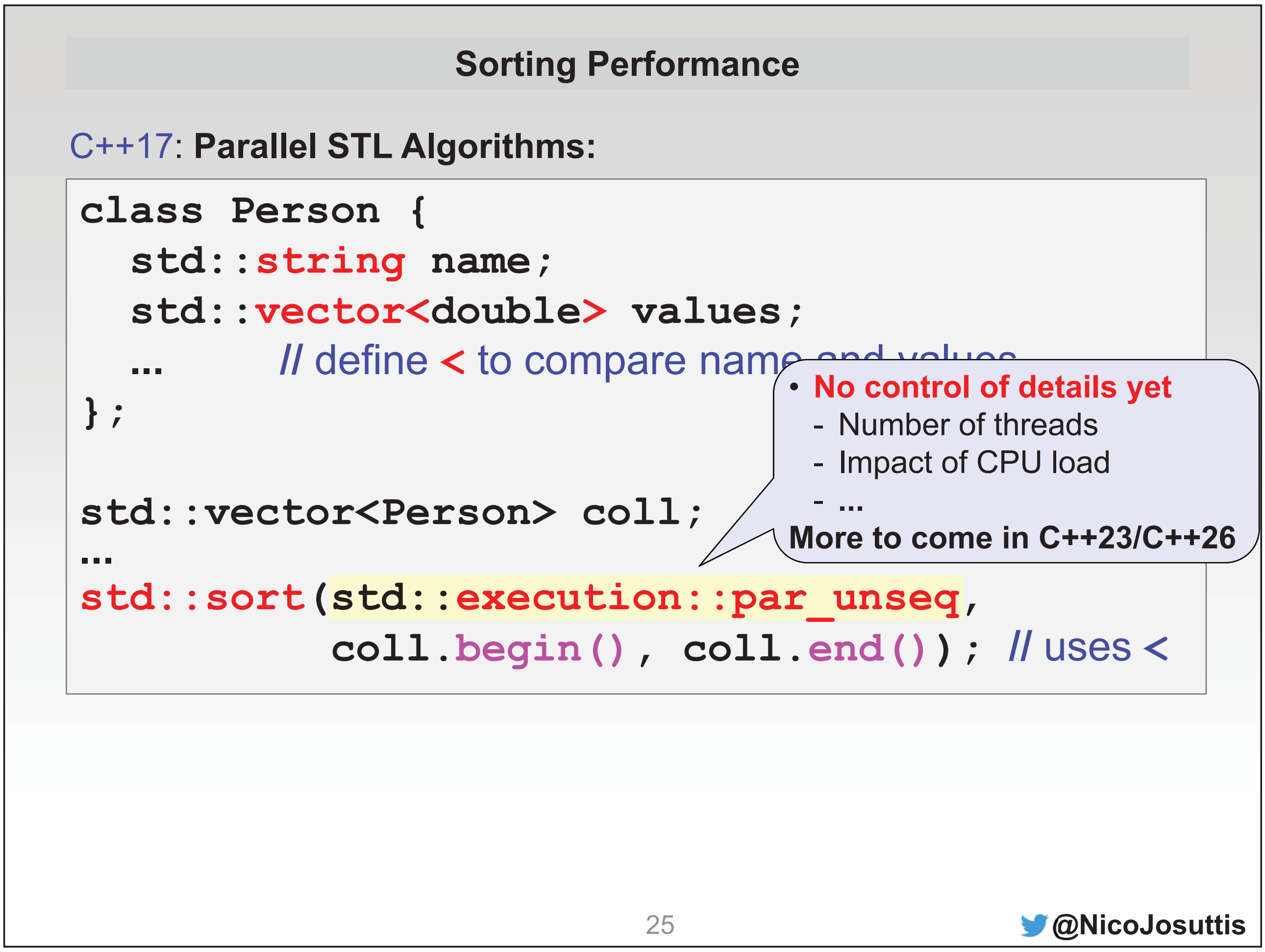
Sequential execution:

Parallel sequenced execution:

multiple threads

Parallel unsequenced execution:

SIMD, vectorization



Sorting Performance

C++17: **Parallel STL Algorithms:**

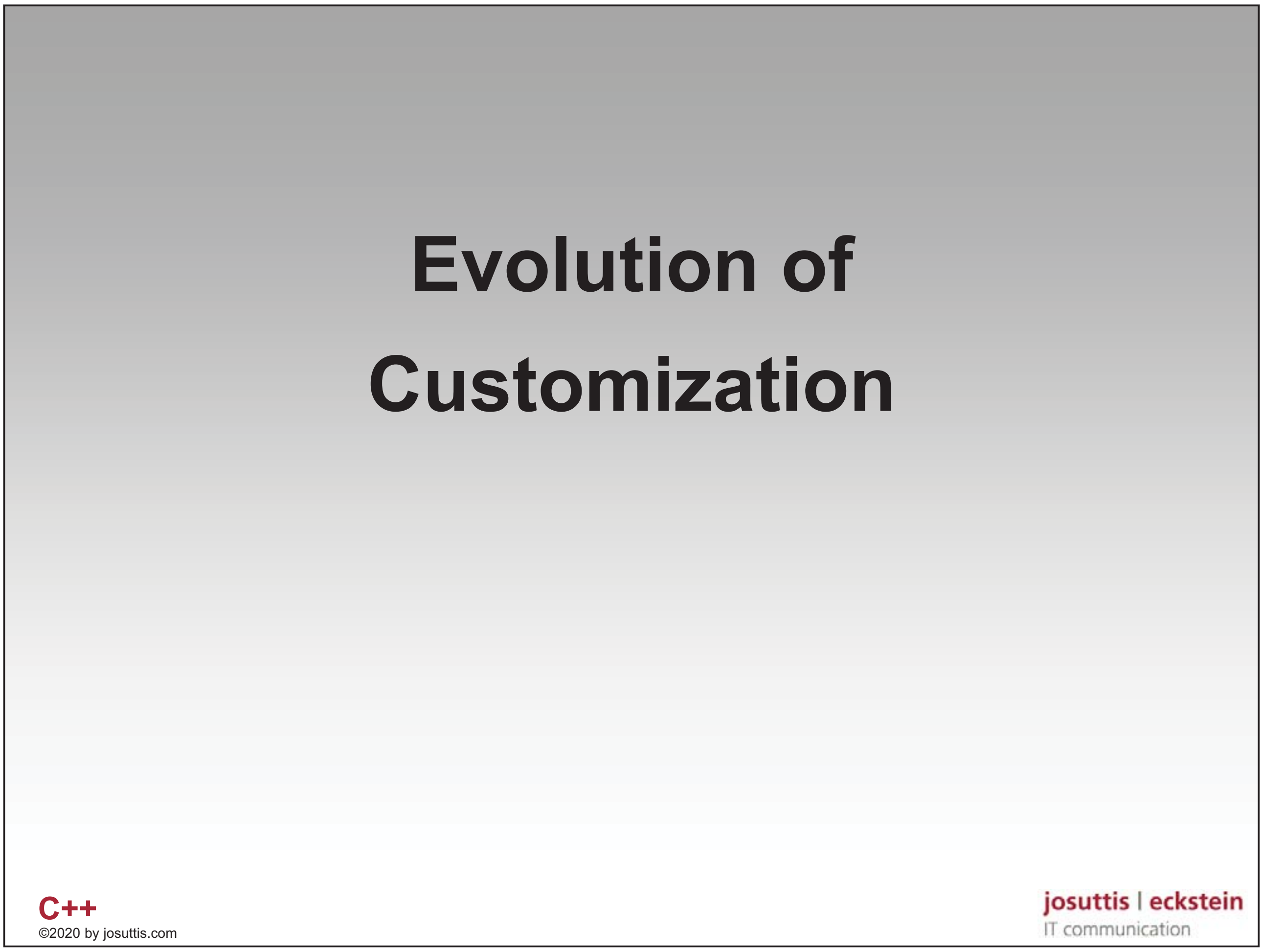
```
class Person {
  std::string name;
  std::vector<double> values;
  ...      // define < to compare name and values
};

std::vector<Person> coll;
...
std::sort(std::execution::par_unseq,
          coll.begin(), coll.end()); // uses <
```

- **No control of details yet**
  - Number of threads
  - Impact of CPU load
  - ...

More to come in C++23/C++26

25

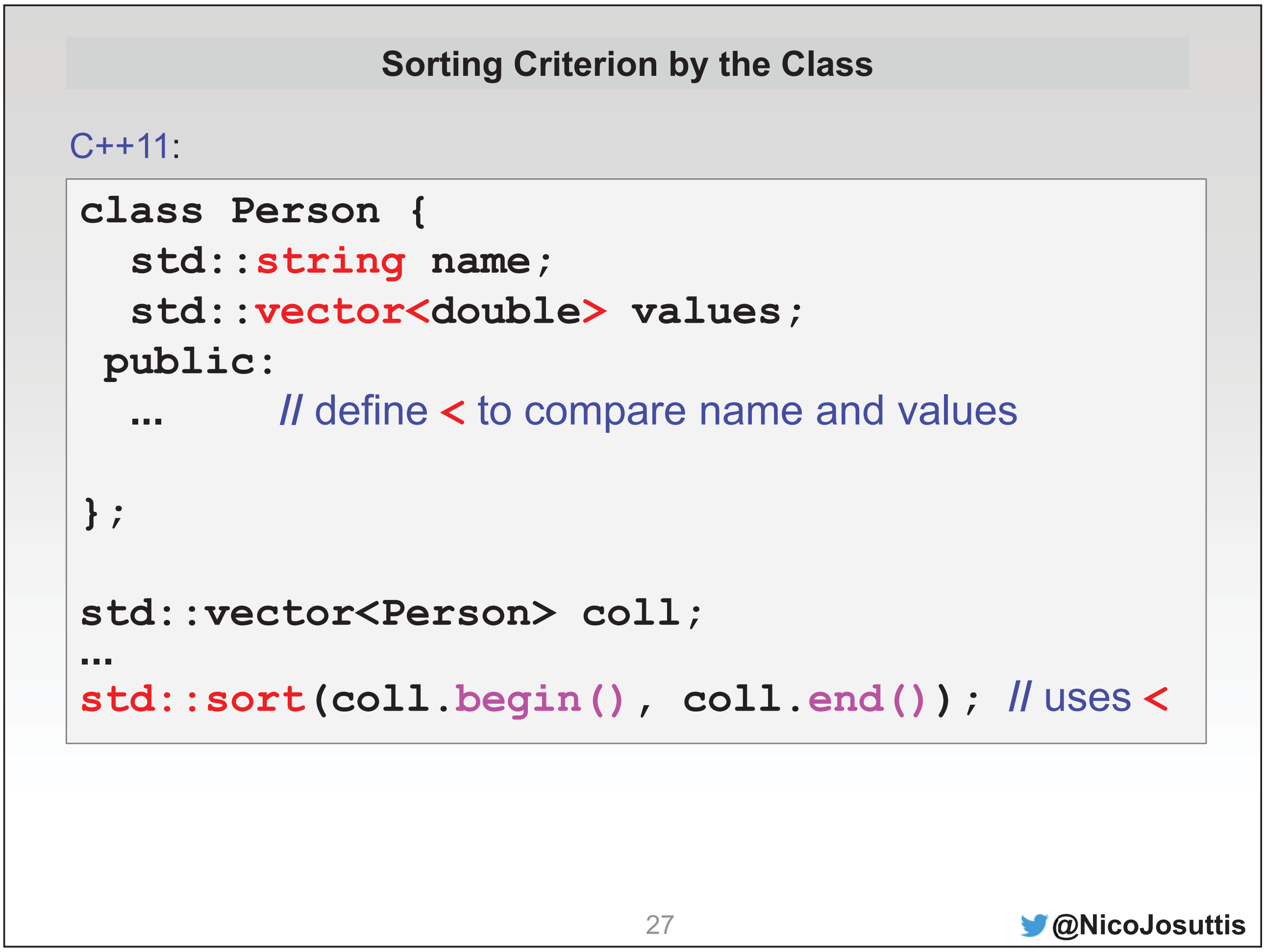
@NicoJosuttis



Evolution of Customization

C++
©2020 by josuttis.com

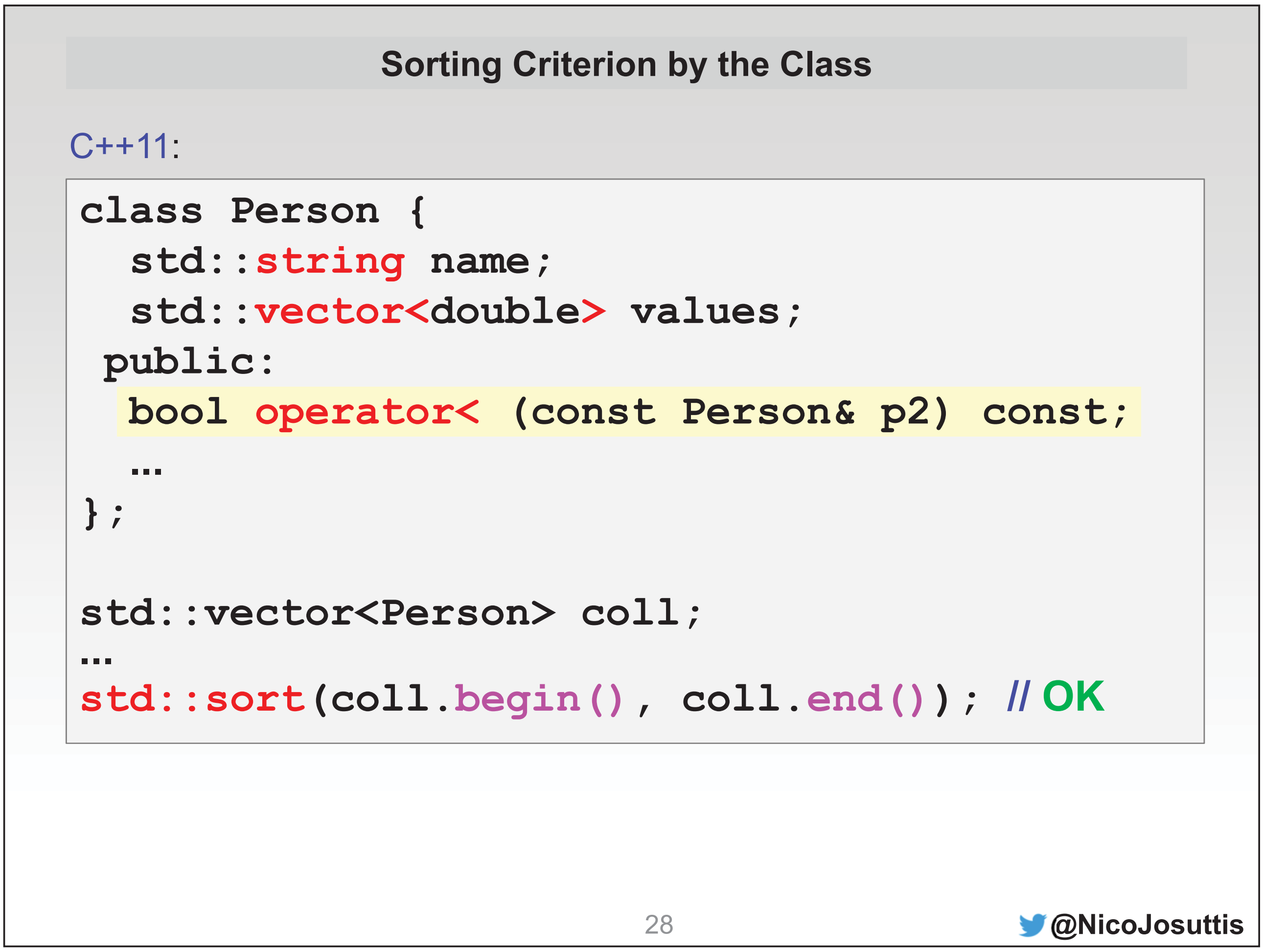
josuttis | eckstein
IT communication



Sorting Criterion by the Class

C++11:

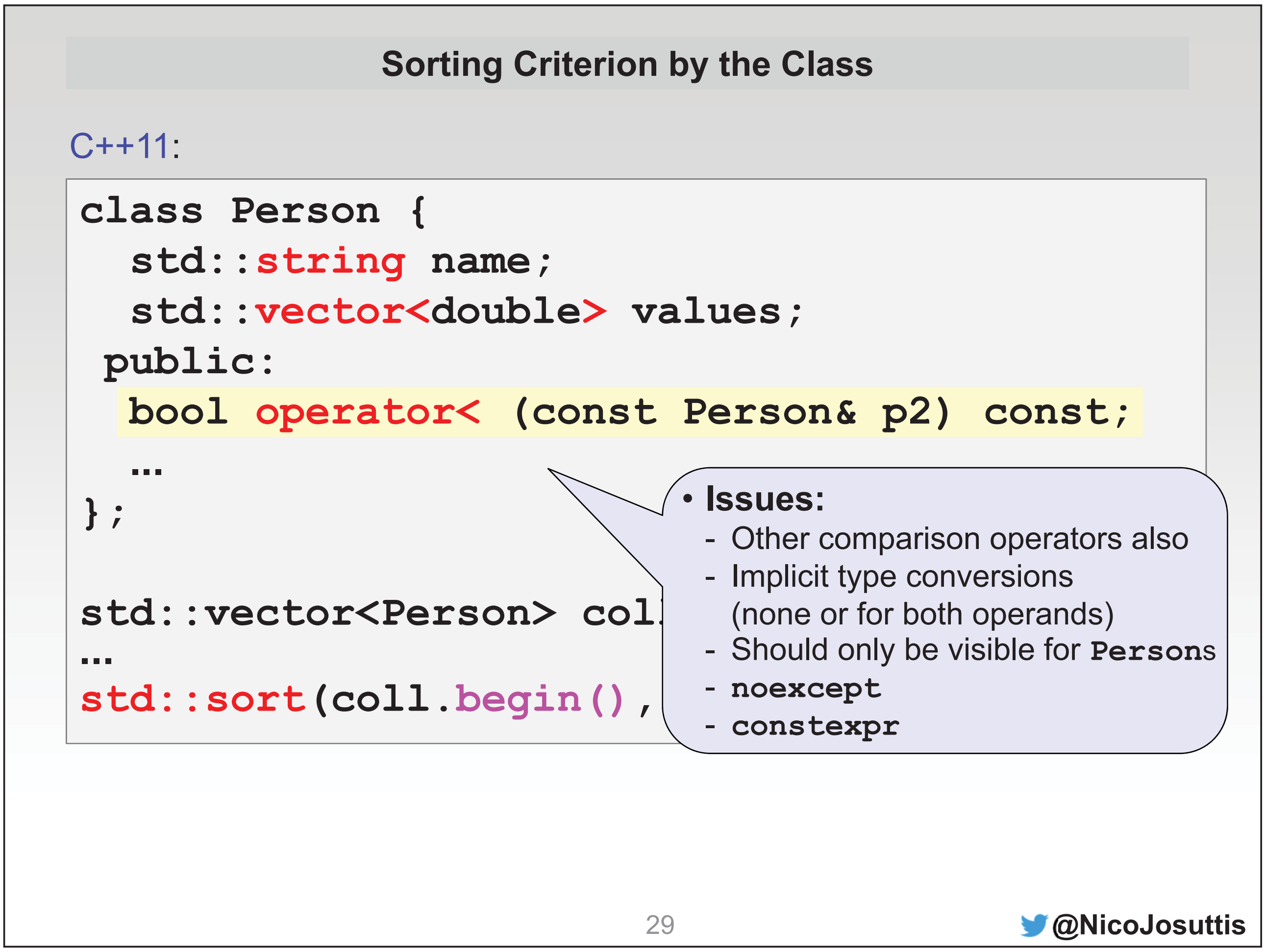
```
class Person {
   std::string name;
   std::vector<double> values;
  public:
   ...      // define < to compare name and values

};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // uses <
```



Sorting Criterion by the Class

C++11:

```
class Person {
   std::string name;
   std::vector<double> values;
  public:
   bool operator< (const Person& p2) const;
   ...
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // OK
```

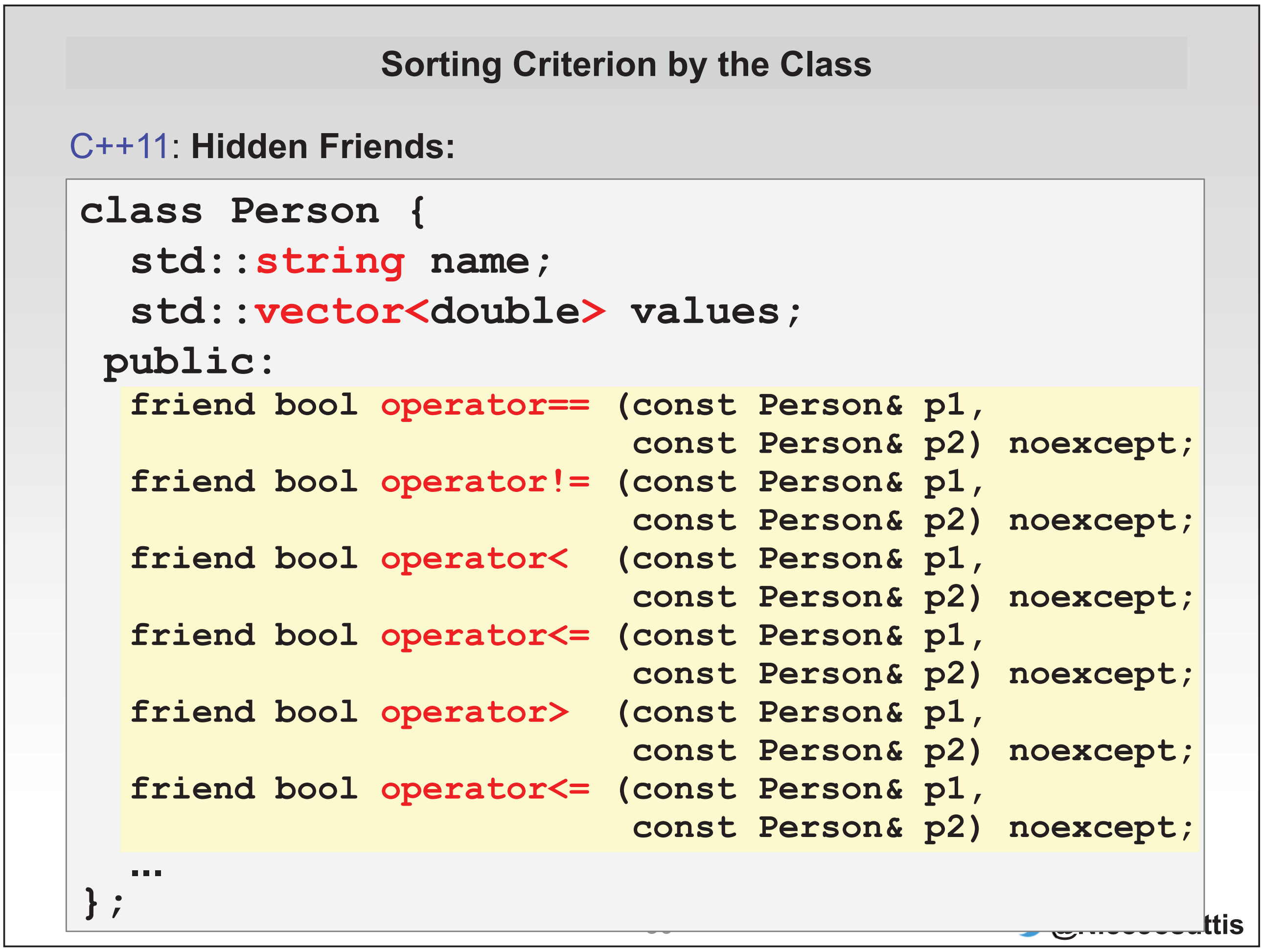


Sorting Criterion by the Class

C++11:

```
class Person {
   std::string name;
   std::vector<double> values;
  public:
   bool operator< (const Person& p2) const;
   ...
};

std::vector<Person> col
...
std::sort(coll.begin(),
```

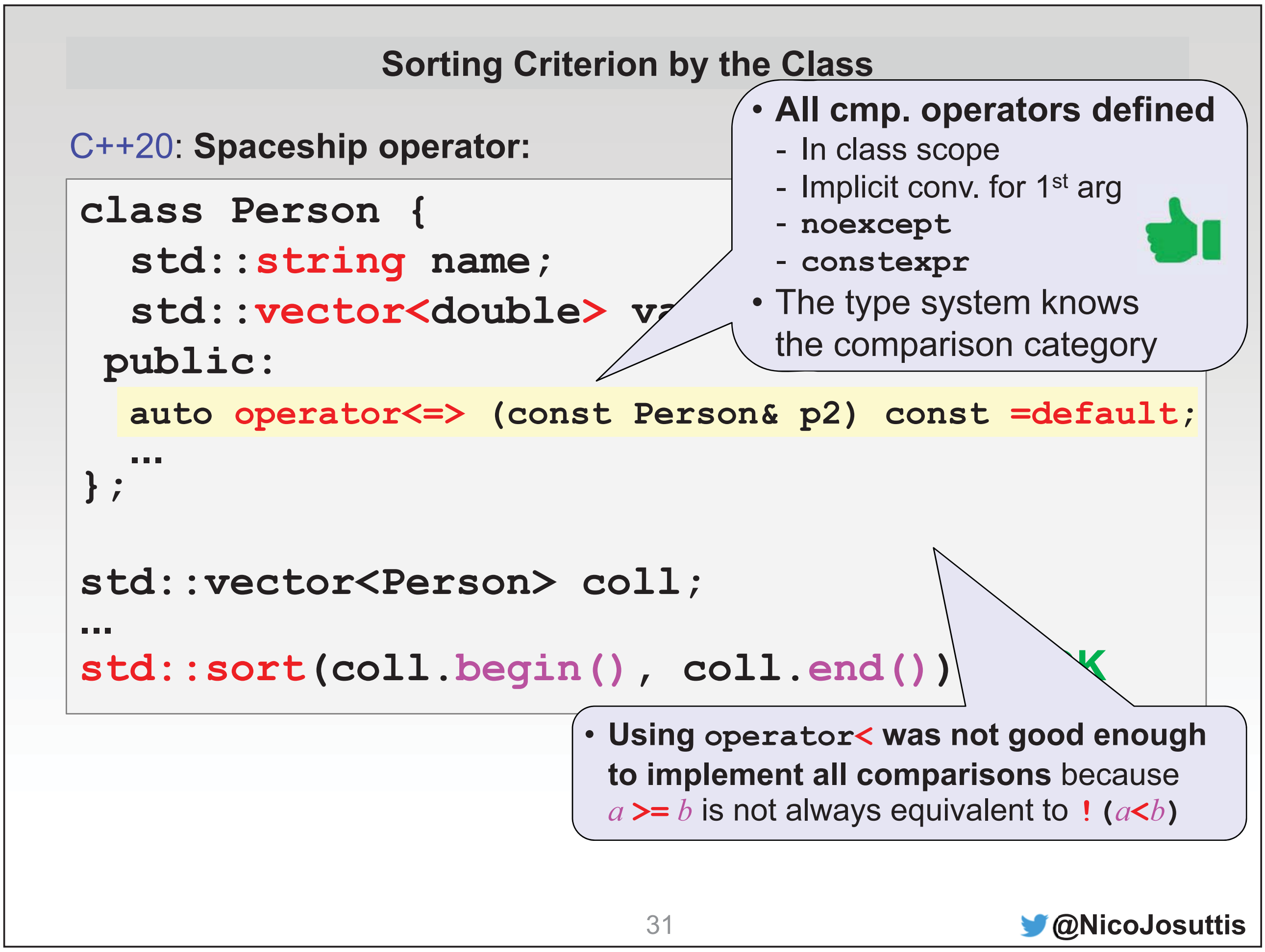
- **Issues:**
  - Other comparison operators also
  - Implicit type conversions (none or for both operands)
  - Should only be visible for `Persons`
  - `noexcept`
  - `constexpr`



Sorting Criterion by the Class

C++11: **Hidden Friends:**

```
class Person {
   std::string name;
   std::vector<double> values;
  public:
   friend bool operator== (const Person& p1,
                            const Person& p2) noexcept;
   friend bool operator!= (const Person& p1,
                            const Person& p2) noexcept;
   friend bool operator<  (const Person& p1,
                            const Person& p2) noexcept;
   friend bool operator<= (const Person& p1,
                            const Person& p2) noexcept;
   friend bool operator>  (const Person& p1,
                            const Person& p2) noexcept;
   friend bool operator<= (const Person& p1,
                            const Person& p2) noexcept;
   ...
};
```



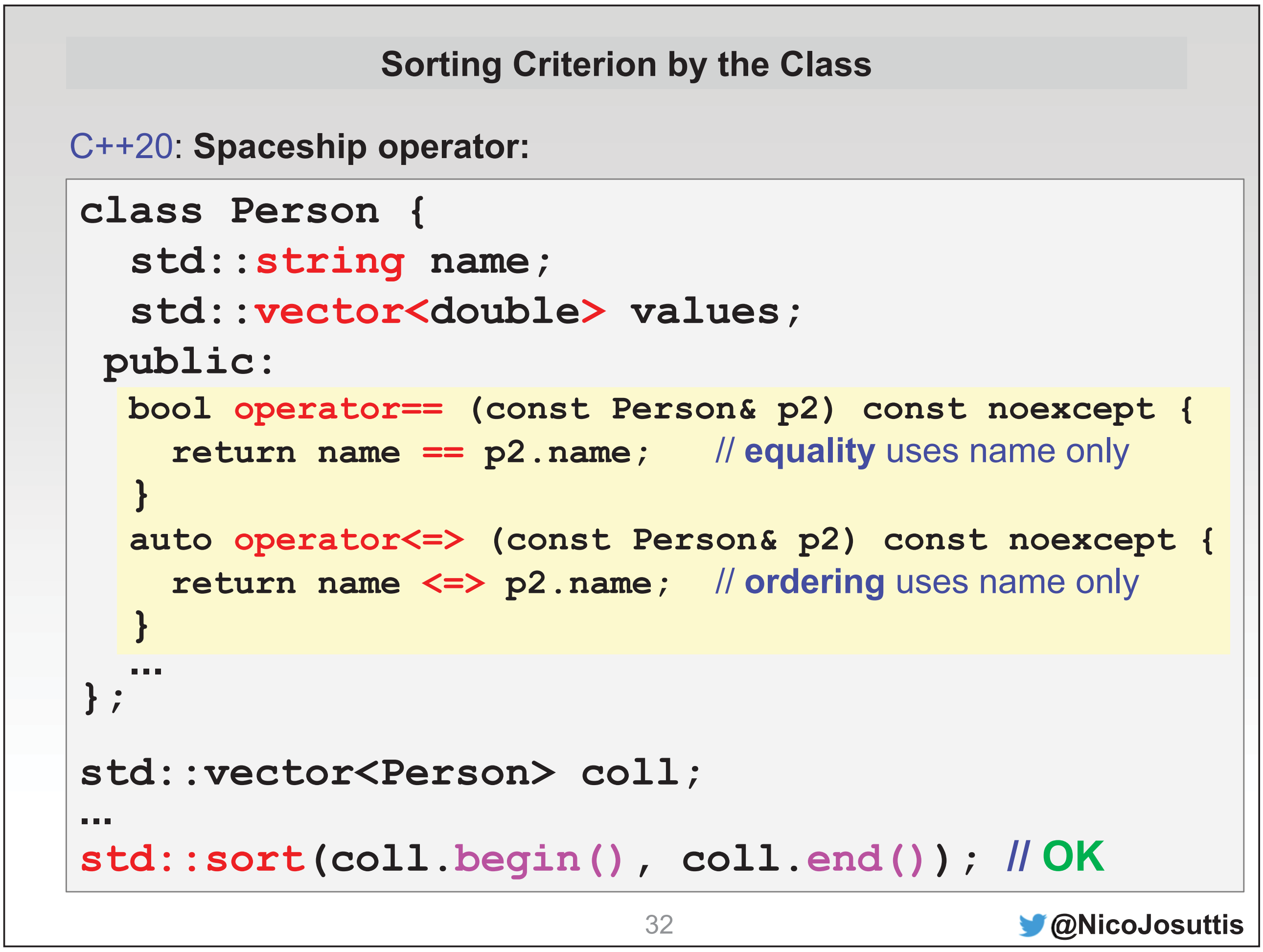
Sorting Criterion by the Class

C++20: **Spaceship operator:**

```
class Person {
  std::string name;
  std::vector<double> va
 public:
  auto operator<=> (const Person& p2) const =default;
  ...
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end())          K
```

- **All cmp. operators defined**
  - In class scope
  - Implicit conv. for 1st arg
  - `noexcept`
  - `constexpr`
- The type system knows the comparison category

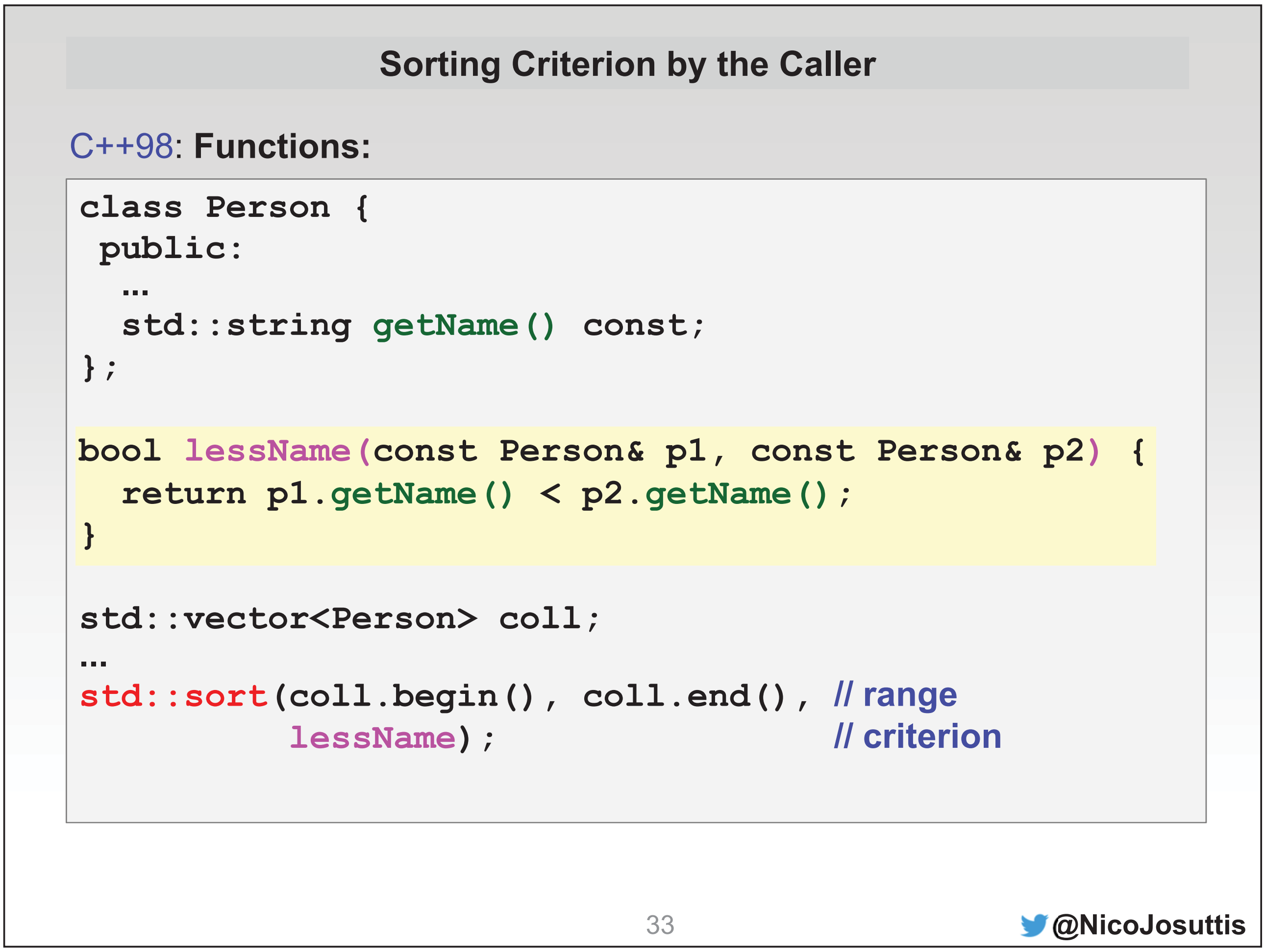
- **Using `operator<` was not good enough to implement all comparisons** because *a* >= *b* is not always equivalent to ! (*a*<*b*)



Sorting Criterion by the Class

C++20: **Spaceship operator:**

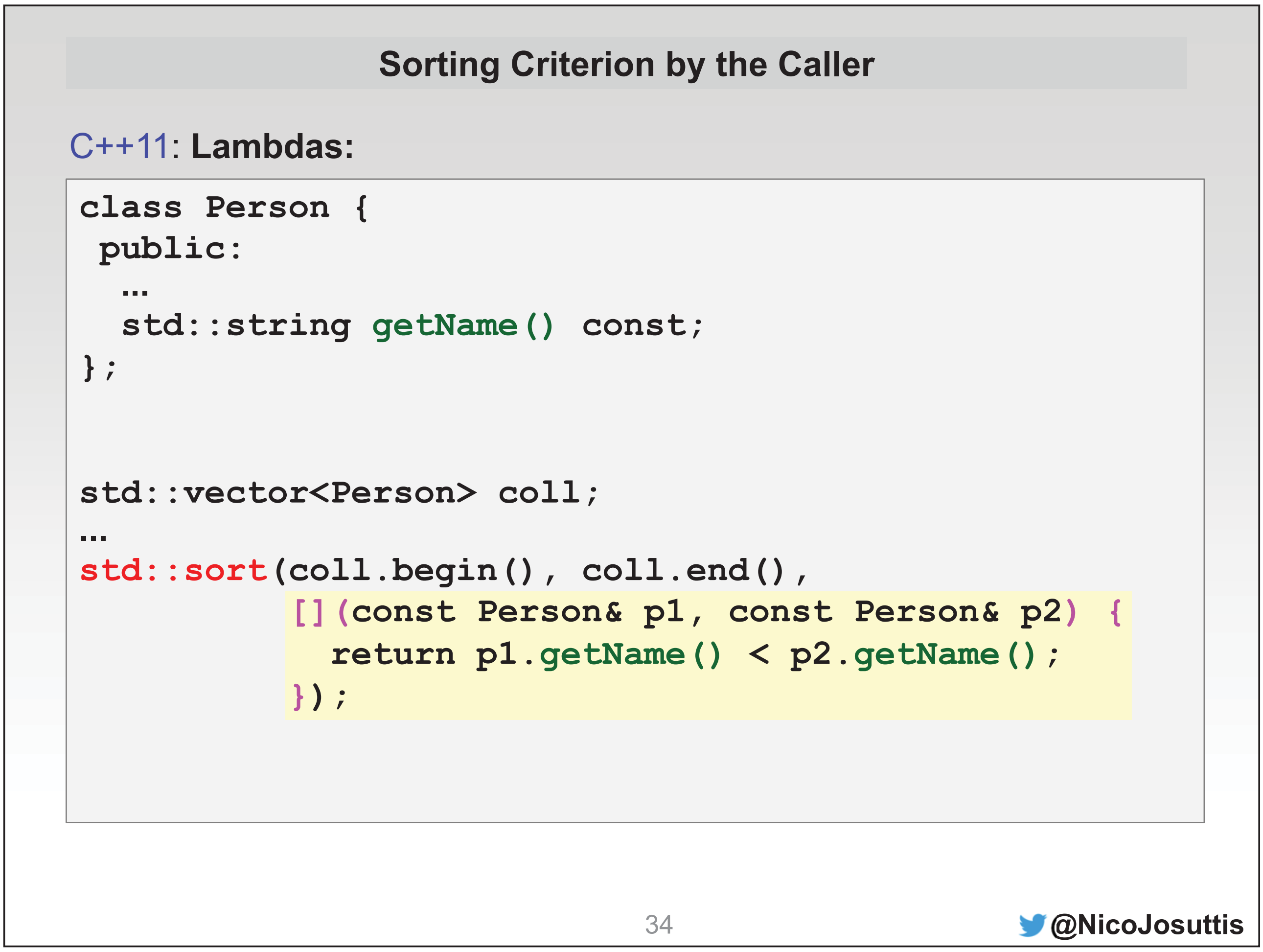
```
class Person {
  std::string name;
  std::vector<double> values;
 public:
  bool operator== (const Person& p2) const noexcept {
    return name == p2.name;    // equality uses name only
  }
  auto operator<=> (const Person& p2) const noexcept {
    return name <=> p2.name;   // ordering uses name only
  }
  ...
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end()); // OK
```



Sorting Criterion by the Caller

C++98: **Functions:**

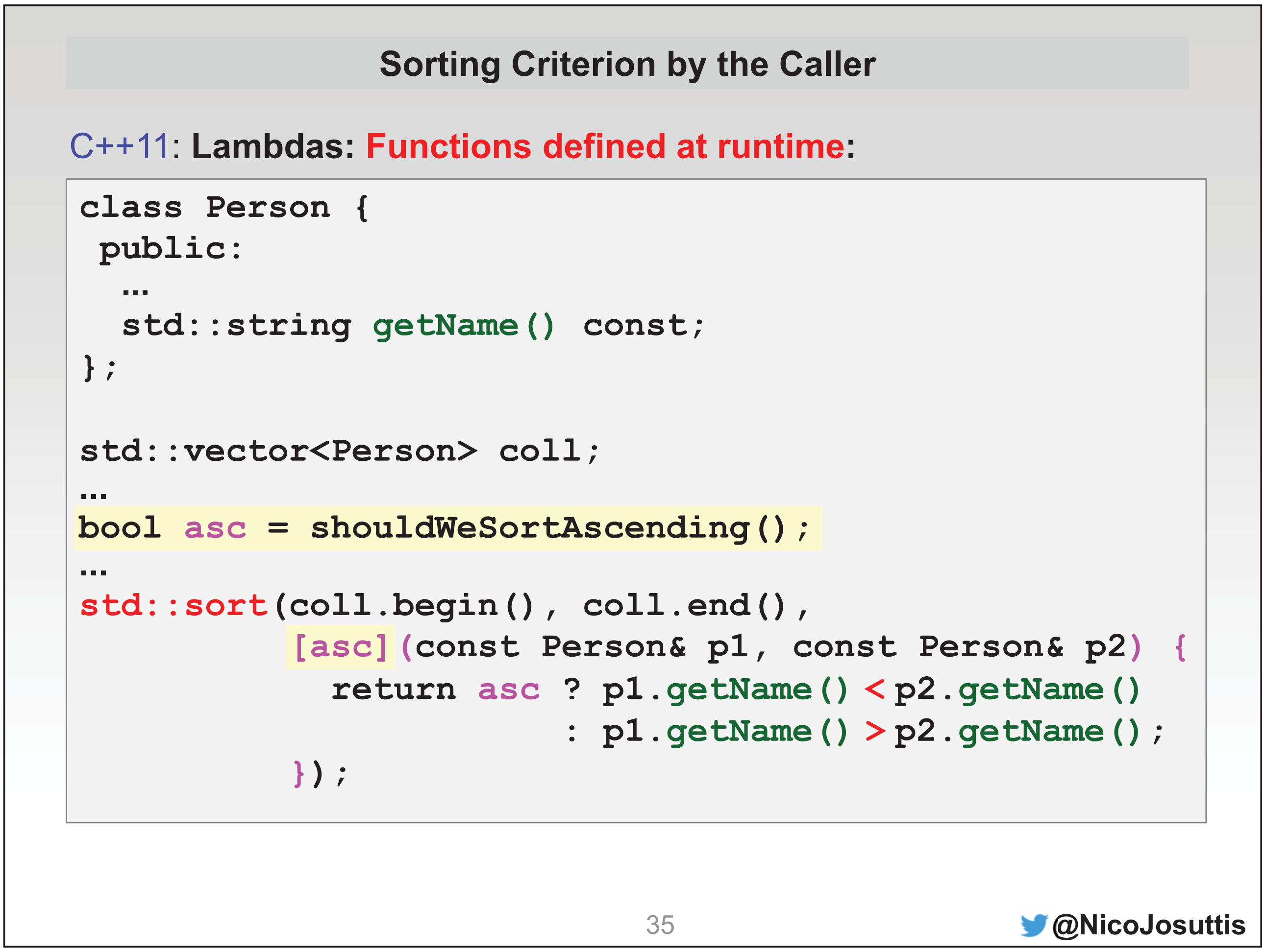
```
class Person {
 public:
  ...
  std::string getName() const;
};

bool lessName(const Person& p1, const Person& p2) {
  return p1.getName() < p2.getName();
}

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end(),  // range
          lessName);                 // criterion
```



Sorting Criterion by the Caller

C++11: **Lambdas:**

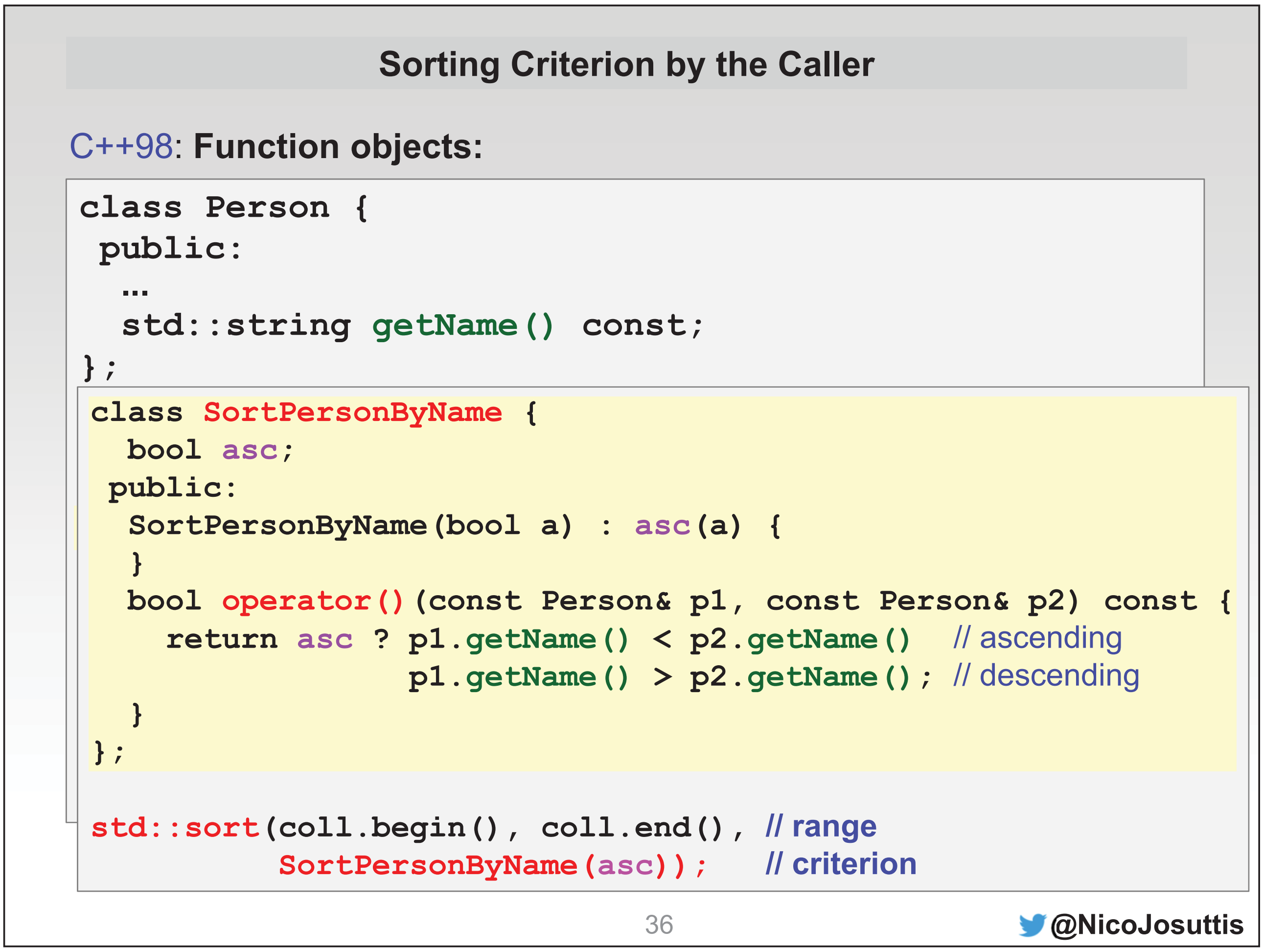
```
class Person {
 public:
  ...
  std::string getName() const;
};


std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end(),
          [](const Person& p1, const Person& p2) {
            return p1.getName() < p2.getName();
          });
```



Sorting Criterion by the Caller

C++11: **Lambdas: Functions defined at runtime:**

```
class Person {
 public:
  ...
  std::string getName() const;
};

std::vector<Person> coll;
...
bool asc = shouldWeSortAscending();
...
std::sort(coll.begin(), coll.end(),
          [asc](const Person& p1, const Person& p2) {
            return asc ? p1.getName()<p2.getName()
                       : p1.getName()>p2.getName();
          });
```

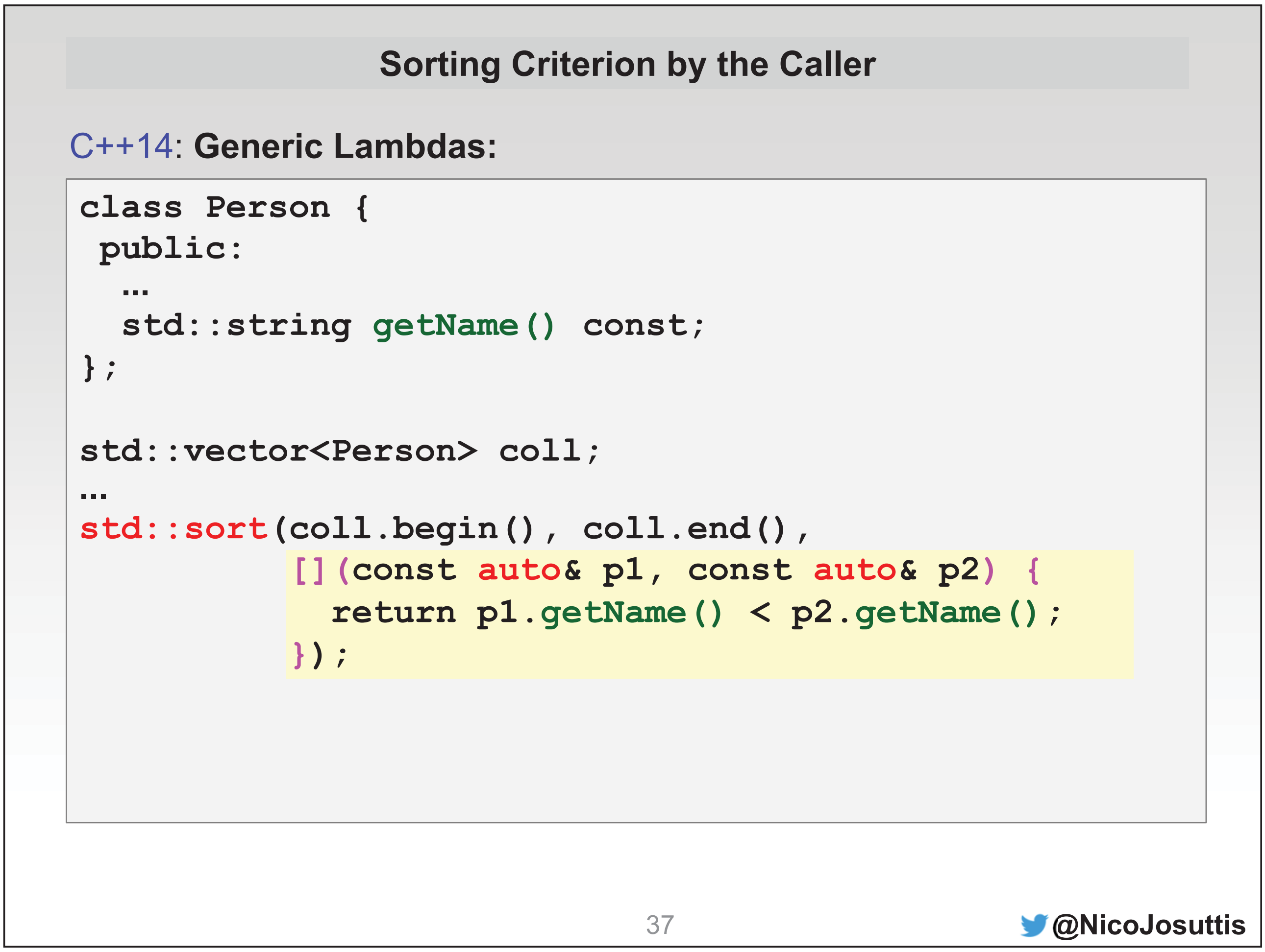


Sorting Criterion by the Caller

C++98: **Function objects:**

```
class Person {
 public:
  ...
  std::string getName() const;
};
```

```
class SortPersonByName {
  bool asc;
 public:
  SortPersonByName(bool a) : asc(a) {
  }
  bool operator()(const Person& p1, const Person& p2) const {
    return asc ? p1.getName() < p2.getName()   // ascending
               : p1.getName() > p2.getName();  // descending
  }
};

std::sort(coll.begin(), coll.end(),   // range
          SortPersonByName(asc));     // criterion
```



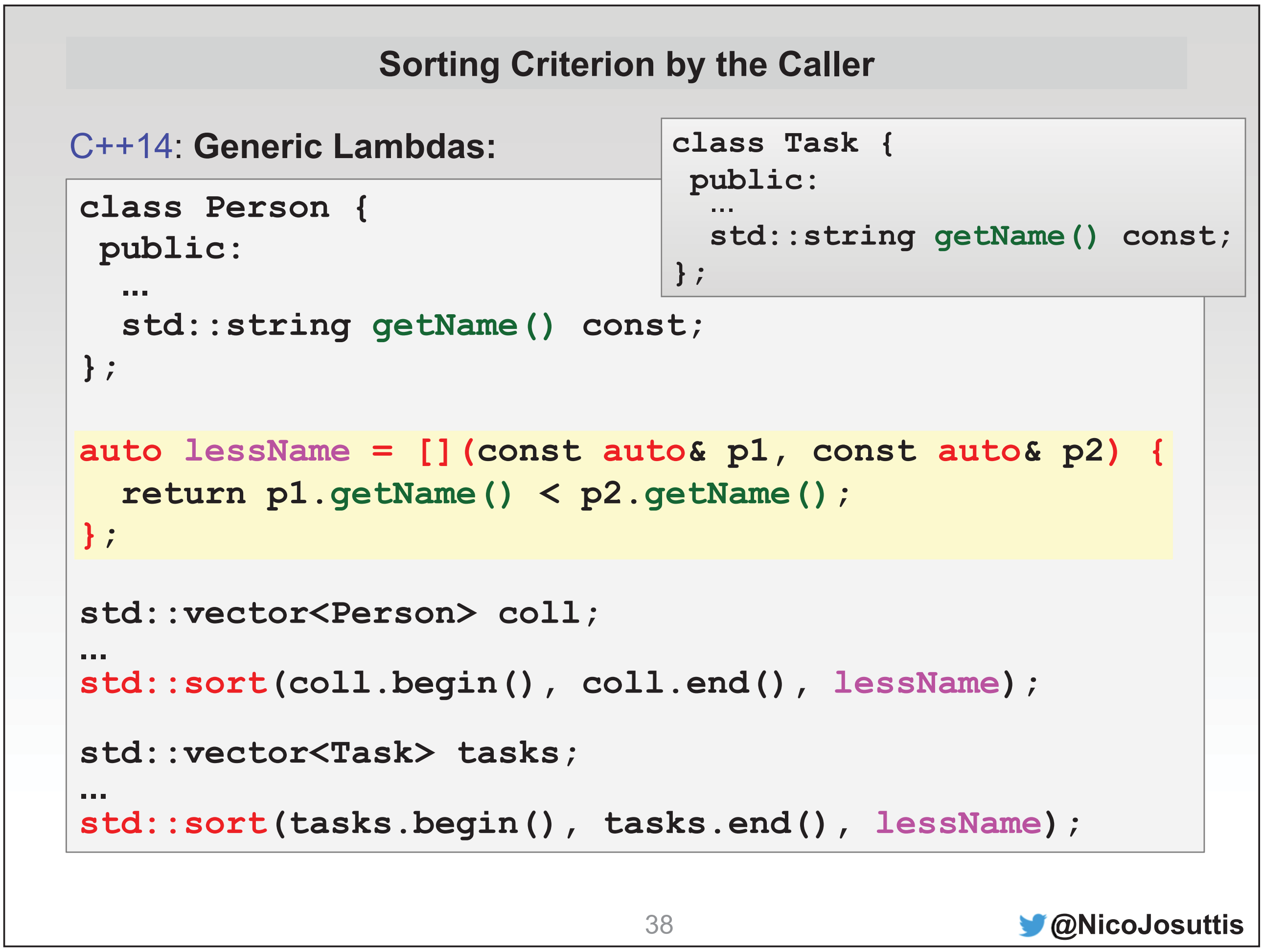
Sorting Criterion by the Caller

C++14: **Generic Lambdas:**

```
class Person {
 public:
  …
  std::string getName() const;
};

std::vector<Person> coll;
…
std::sort(coll.begin(), coll.end(),
          [](const auto& p1, const auto& p2) {
            return p1.getName() < p2.getName();
          });
```

37

@NicoJosuttis



Sorting Criterion by the Caller

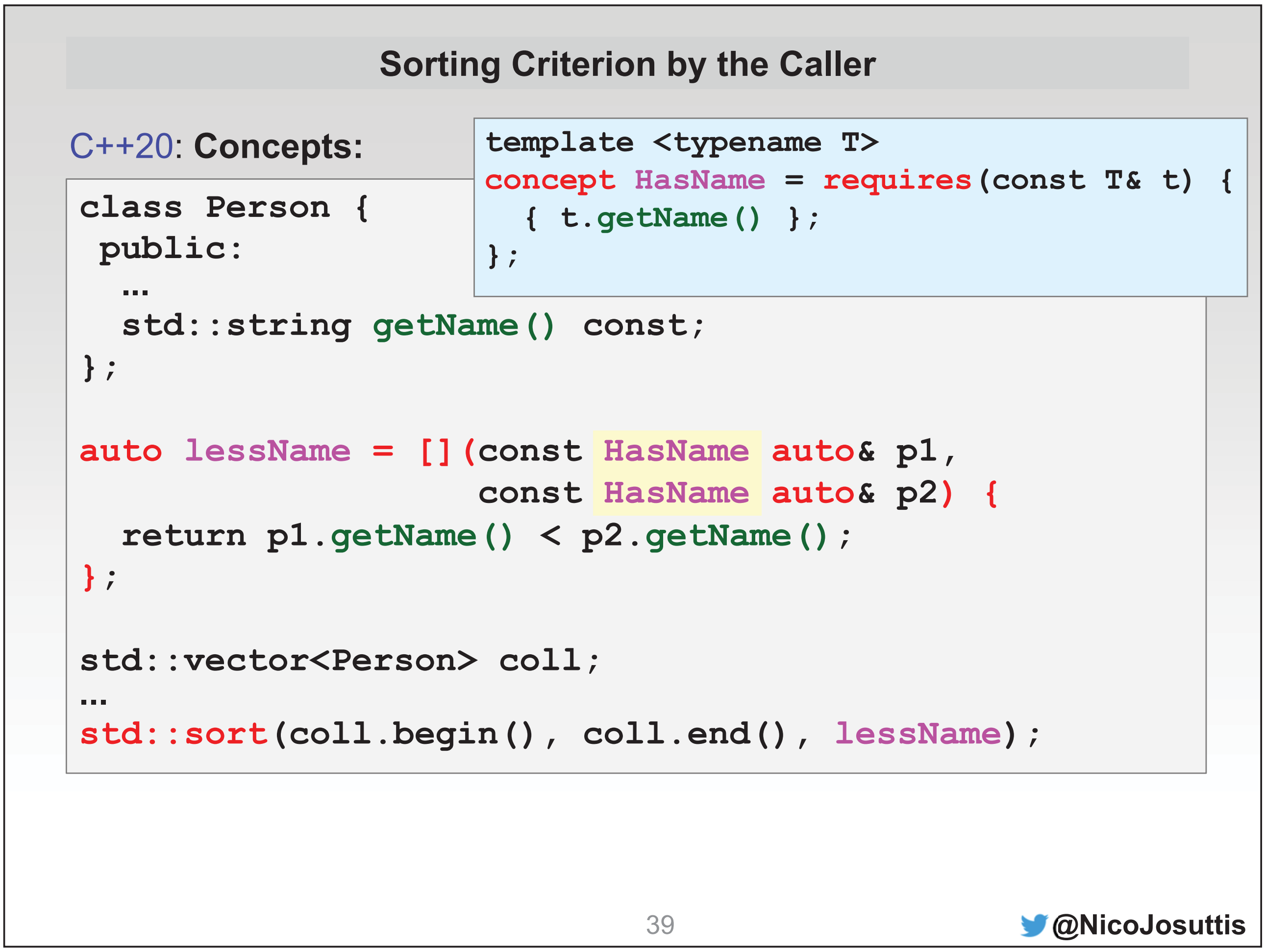
C++14: **Generic Lambdas:**

```
class Task {
 public:
  …
  std::string getName() const;
};
```

```
class Person {
 public:
  …
  std::string getName() const;
};

auto lessName = [](const auto& p1, const auto& p2) {
  return p1.getName() < p2.getName();
};

std::vector<Person> coll;
…
std::sort(coll.begin(), coll.end(), lessName);

std::vector<Task> tasks;
…
std::sort(tasks.begin(), tasks.end(), lessName);
```

38

@NicoJosuttis



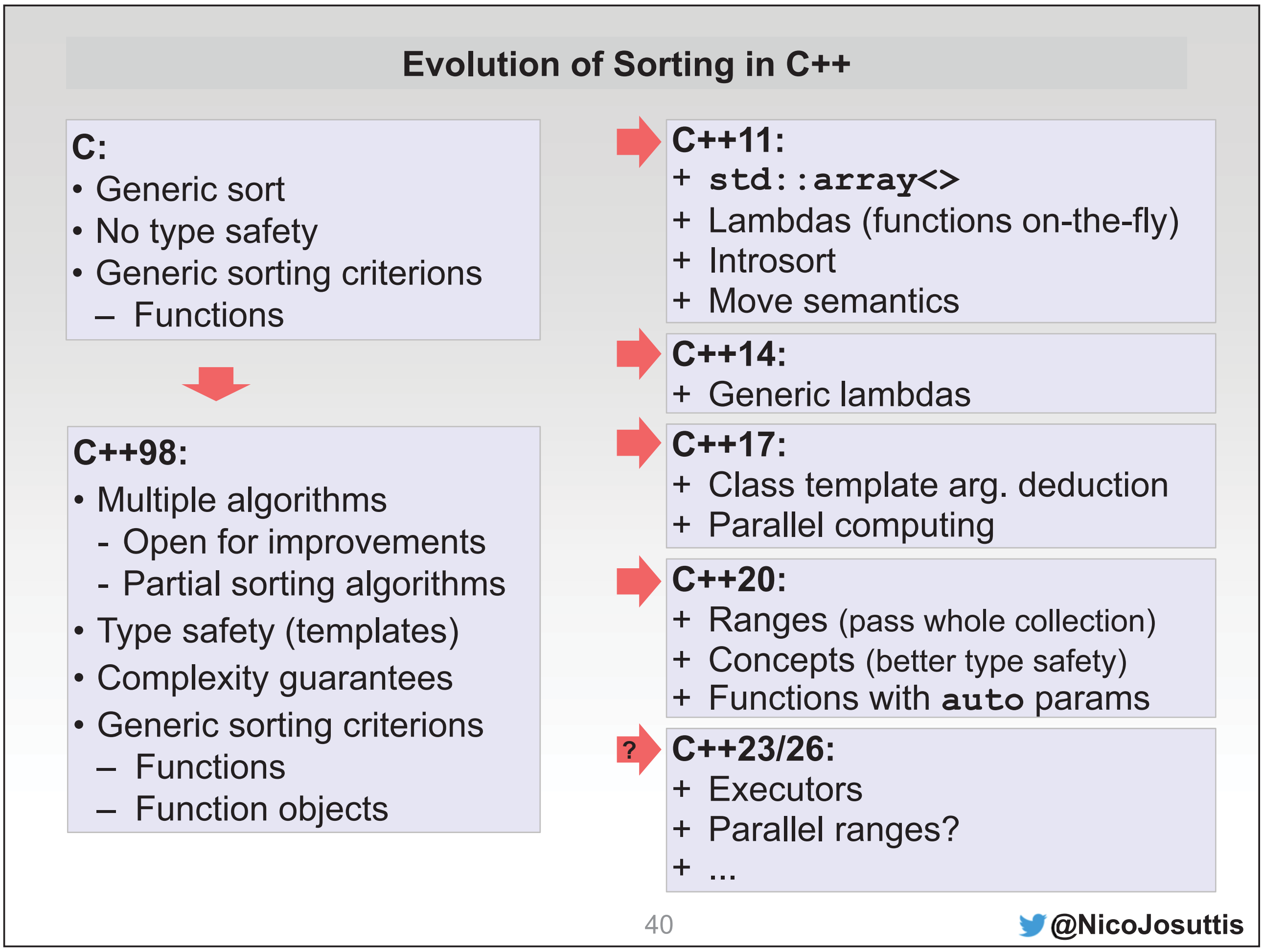
Sorting Criterion by the Caller

C++20: **Concepts:**

```
template <typename T>
concept HasName = requires(const T& t) {
  { t.getName() };
};
```

```
class Person {
 public:
  ...
  std::string getName() const;
};

auto lessName = [](const HasName auto& p1,
                   const HasName auto& p2) {
  return p1.getName() < p2.getName();
};

std::vector<Person> coll;
...
std::sort(coll.begin(), coll.end(), lessName);
```

39 @NicoJosuttis



Evolution of Sorting in C++

C:
- Generic sort
- No type safety
- Generic sorting criterions
  – Functions

C++98:
- Multiple algorithms
  - Open for improvements
  - Partial sorting algorithms
- Type safety (templates)
- Complexity guarantees
- Generic sorting criterions
  – Functions
  – Function objects

C++11:
+ `std::array<>`
+ Lambdas (functions on-the-fly)
+ Introsort
+ Move semantics

C++14:
+ Generic lambdas

C++17:
+ Class template arg. deduction
+ Parallel computing

C++20:
+ Ranges (pass whole collection)
+ Concepts (better type safety)
+ Functions with `auto` params

? **C++23/26:**
+ Executors
+ Parallel ranges?
+ ...

40 @NicoJosuttis

Lessons Learned

- **Programming evolves**
- **Contexts evolve**
- **C++ evolves**
  - Performance
  - Usability
- **But we have to be backward compatible**
  - for almost 50 years
- **We make mistakes**
  - Don't complain, take care
- **We are getting better**
- **Tools improve**

It's all your fault, because you didn't help us to make it better 😉

Take Care

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis

Download slides at:
josuttis.com/download/qt