Back To Basics
Algorithms

KLAUS IGLBERGER

C++ Trainer/Consultant

Author of “C++ Software Design”

Chair of the CppCon Software Design track

(Co-)Organizer of the Munich C++ user group

Email: klaus.iglberger@gmx.de

Klaus Iglberger

It Began with a Proposal …

The Standard Template Library

Alexander Stepanov

Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mt. View, CA 94043
stepanov@mti.sgi.com

Meng Lee

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
lee@hpl.hp.com

October 31, 1995

Alexander Stepanov

The STL in a Nutshell

The main ingredient for the STL are …

- **Containers**: Implementations of the common data collections
- **Algorithms**: work on the data contained in containers
- **Iterators**: The glue between containers and algorithms

The STL in a Nutshell

The STL in a Nutshell

The main ingredient for the STL are …

- **Containers**: Implementations of the common data collections
- **Algorithms**: work on the data contained in containers
- **Iterators**: The glue between containers and algorithms

Back To Basics - Iterators on Tuesday at 2pm

Let’s have some fun with

Algorithms

and …

Thanks to my former colleague Daniel Pfeifer for the original idea!

Picture taken from Wikipedia. https://en.wikipedia.org/wiki/The_Simpsons#/media/File:Simpsons_cast.png

The Cast for this Talk

Lisa

The algorithm expert

Maggie

The always interested C++ enthusiast

Bart

The algorithm ignorant skeptic

Fun with the Simpsons


```cpp
#include <cstdlib>
#include <string>
#include <vector>


struct Person
{
   std::string firstname;
   std::string lastname;
   int age;
};



int main()
{
   std::vector<Person> table =
      { Person{ "Homer",      "Simpson",    38 }
      , Person{ "Marge",      "Simpson",    34 }
      , Person{ "Bart",       "Simpson",    10 }
      , Person{ "Lisa",       "Simpson",     8 }
      , Person{ "Maggie",     "Simpson",     1 }
      , Person{ "Hans",       "Moleman",    33 }
      , Person{ "Ralph",      "Wiggum",      8 }
      , Person{ "Milhouse",   "Van Houten", 10 }
      , Person{ "Ned",        "Flanders",   60 }
```

Fun with the Simpsons


```
int main()
{
    std::vector<Person> table =
        { Person{ "Homer",      "Simpson",    38 }
        , Person{ "Marge",      "Simpson",    34 }
        , Person{ "Bart",       "Simpson",    10 }
        , Person{ "Lisa",       "Simpson",     8 }
        , Person{ "Maggie",     "Simpson",     1 }
        , Person{ "Hans",       "Moleman",    33 }
        , Person{ "Ralph",      "Wiggum",      8 }
        , Person{ "Milhouse",   "Van Houten", 10 }
        , Person{ "Ned",        "Flanders",   60 }
        , Person{ "Jeff",       "Albertson",  45 }
        , Person{ "Montgomery", "Burns",     104 }
        , /* many more Simpsons characters */ };

    // ...

    return EXIT_SUCCESS;
}
```

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Find the youngest person

    // ...
```

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Find the youngest person
    if( begin(table) == end(table) ) {
        std::cout << "Empty table, no youngest person!\n";
    }
    else {
        auto smallest = begin(table);

        for( auto pos=begin(table)+1; pos!=end(table); ++pos )
        {
            if( pos->age < smallest->age ) {
                smallest = pos;
            }
        }

        std::cout << "Youngest person = " << smallest->firstname
                  << " " << smallest->lastname << '\n';
    }

    // ...
```

Maggie Disapproves Bart's Solution!

```cpp
int main()
{
   std::vector<Person> table = /* ... */;

   // Find the youngest person
   if( begin(table) == end(table) ) {
      std::cout << "Empty table, no youngest person!\n";
   }
   else {
      auto smallest = begin(table);

      for( auto pos=begin(table)+1; pos!=end(table); ++pos )
      {
         if( pos->age < smallest->age ) {
            smallest = pos;
         }
      }

      std::cout << "Youngest person = " << smallest->firstname
                << " " << smallest->lastname << '\n';
   }

   // ...
```

This comment is a clear indication that something is wrong

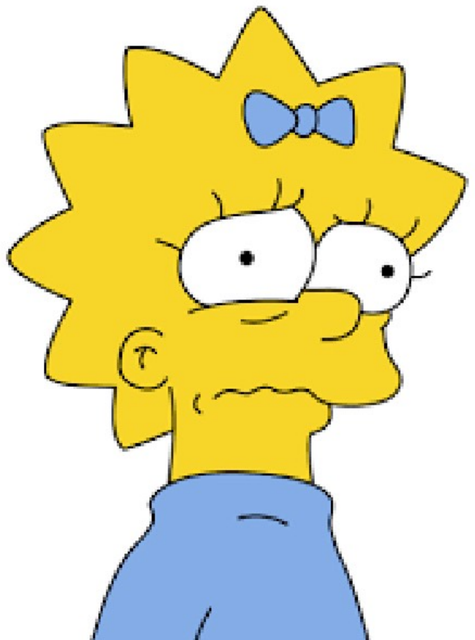
Hard to read

Hard to understand

Verbose

Easy to get wrong

Not reusable



Maggie Would Use An Algorithm

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

- Freestanding implementations
- ASCII chart

Language
- Basic concepts
- Keywords
- Preprocessor
- Expressions
- Declarations
- Initialization
- Functions
- Statements
- Classes
- Overloading
- Templates
- Exceptions

Standard library (headers)

Named requirements

Feature test macros (C++20)

Metaprogramming library (C++11)
- Type traits – `ratio`
- `integer_sequence` (C++14)

General utilities library
- Function objects – `hash` (C++11)
- Swap – Type operations (C++11)
- Integer comparison (C++20)
- `pair` – `tuple` (C++11)
- `optional` (C++17)
- `expected` (C++23)
- `variant` (C++17) – `any` (C++17)
- String conversions (C++17)
- Formatting (C++20)
- `bitset` – Bit manipulation (C++20)

Strings library
- `basic_string` – `char_traits`
- `basic_string_view` (C++17)
- Null-terminated strings:

Iterators library

Ranges library (C++20)

Algorithms library
- Execution policies (C++17)
- Constrained algorithms (C++20)

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Numeric algorithms
- Pseudo-random number generation
- Floating-point environment (C++11)
- `complex` – `valarray`

Date and time library
- Calendar (C++20) – Time zone (C++20)

Localizations library
- `locale` – Character classification

Input/output library

Algorithms library

The algorithms library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Note that a range is defined as [`first` , `last`) where `last` refers to the element *past* the last element to inspect or modify.

Constrained algorithms

C++20 provides constrained versions of most algorithms in the namespace `std::ranges`. In these algorithms, a range can be specified as either an iterator-sentinel pair or as a single range argument, and projections and pointer-to-member callables are supported. Additionally, the return types of most algorithms have been changed to return all potentially useful information computed during the execution of the algorithm. (since C++2

```
std::vector<int> v {7, 1, 4, 0, -1};
std::ranges::sort(v); // constrained algorithm
```

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several execution policies, and the library provides corresponding execution policy types and objects. Users

Minimum/maximum operations

Defined in header <algorithm>

max	returns the greater of the given values (function template)
ranges::max (C++20)	returns the greater of the given values (niebloid)
max_element	returns the largest element in a range (function template)
ranges::max_element (C++20)	returns the largest element in a range (niebloid)
min	returns the smaller of the given values (function template)
ranges::min (C++20)	returns the smaller of the given values (niebloid)
min_element	returns the smallest element in a range (function template)
ranges::min_element (C++20)	returns the smallest element in a range (niebloid)
minmax (C++11)	returns the smaller and larger of two elements (function template)
ranges::minmax (C++20)	returns the smaller and larger of two elements (niebloid)
minmax_element (C++11)	returns the smallest and the largest elements in a range (function template)
ranges::minmax_element (C++20)	returns the smallest and the largest elements in a range (niebloid)
clamp (C++17)	clamps a value between a pair of boundary values (function template)
ranges::clamp (C++20)	clamps a value between a pair of boundary values (niebloid)

std::min_element

Defined in header <algorithm>

```
template< class ForwardIt >
ForwardIt min_element( ForwardIt first, ForwardIt last );
```
(1) (until C++17)

```
template< class ForwardIt >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last );
```
(1) (since C++17)

```
template< class ExecutionPolicy, class ForwardIt >
ForwardIt min_element( ExecutionPolicy&& policy,
                       ForwardIt first, ForwardIt last );
```
(2) (since C++17)

```
template< class ForwardIt, class Compare >
ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );
```
(3) (until C++17)

```
template< class ForwardIt, class Compare >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last,
                                 Compare comp );
```
(3) (since C++17)

```
template< class ExecutionPolicy, class ForwardIt, class Compare >
ForwardIt min_element( ExecutionPolicy&& policy,
                       ForwardIt first, ForwardIt last, Compare comp );
```
(4) (since C++17)

Finds the smallest element in the range [`first` , `last`).

1) Elements are compared using operator<.

Introducing std::min_element()


```
int main()
{
   std::vector<Person> table =   /* ... */;

   // ...

   // Find the youngest person
   auto const pos =
      std::min_element( begin(table), end(table) );  Compilation error!




   if( pos != end(table) ) {
      std::cout << "Youngest person = " << pos->firstname
                << " " << pos->lastname << '\n';
   }


   // ...
```

std::min_element

Defined in header <algorithm>

```
template< class ForwardIt >
ForwardIt min_element( ForwardIt first, ForwardIt last );                    (1)  (until C++17)
template< class ForwardIt >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last );          (1)  (since C++17)

template< class ExecutionPolicy, class ForwardIt >
ForwardIt min_element( ExecutionPolicy&& policy,                             (2)  (since C++17)
                       ForwardIt first, ForwardIt last );

template< class ForwardIt, class Compare >
ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );      (3)  (until C++17)
template< class ForwardIt, class Compare >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last,            (3)  (since C++17)
                                 Compare comp );

template< class ExecutionPolicy, class ForwardIt, class Compare >
ForwardIt min_element( ExecutionPolicy&& policy,                             (4)  (since C++17)
                       ForwardIt first, ForwardIt last, Compare comp );
```

Finds the smallest element in the range [`first` , `last`).

1) Elements are compared using operator<.

standard policies, `std::terminate` is called. For any other `ExecutionPolicy`, the behavior is implementation-defined.

- If the algorithm fails to allocate memory, `std::bad_alloc` is thrown.

Possible implementation

min_element (1)

```
template<class ForwardIt>
ForwardIt min_element(ForwardIt first, ForwardIt last)
{
    if (first == last)
        return last;

    ForwardIt smallest = first;
    ++first;

    for (; first != last; ++first)
        if (*first < *smallest)
            smallest = first;

    return smallest;
}
```

min_element (3)

```
template<class ForwardIt, class Compare>
```

std::min_element

Defined in header <algorithm>

```
template< class ForwardIt >
ForwardIt min_element( ForwardIt first, ForwardIt last );                          (1)  (until C++17)
template< class ForwardIt >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last );                     (since C++17)

template< class ExecutionPolicy, class ForwardIt >
ForwardIt min_element( ExecutionPolicy&& policy,                                   (2)  (since C++17)
                       ForwardIt first, ForwardIt last );

template< class ForwardIt, class Compare >
ForwardIt min_element( ForwardIt first, ForwardIt last, Compare comp );            (3)  (until C++17)
template< class ForwardIt, class Compare >
constexpr ForwardIt min_element( ForwardIt first, ForwardIt last,                       (since C++17)
                                 Compare comp );

template< class ExecutionPolicy, class ForwardIt, class Compare >
ForwardIt min_element( ExecutionPolicy&& policy,                                   (4)  (since C++17)
                       ForwardIt first, ForwardIt last, Compare comp );
```

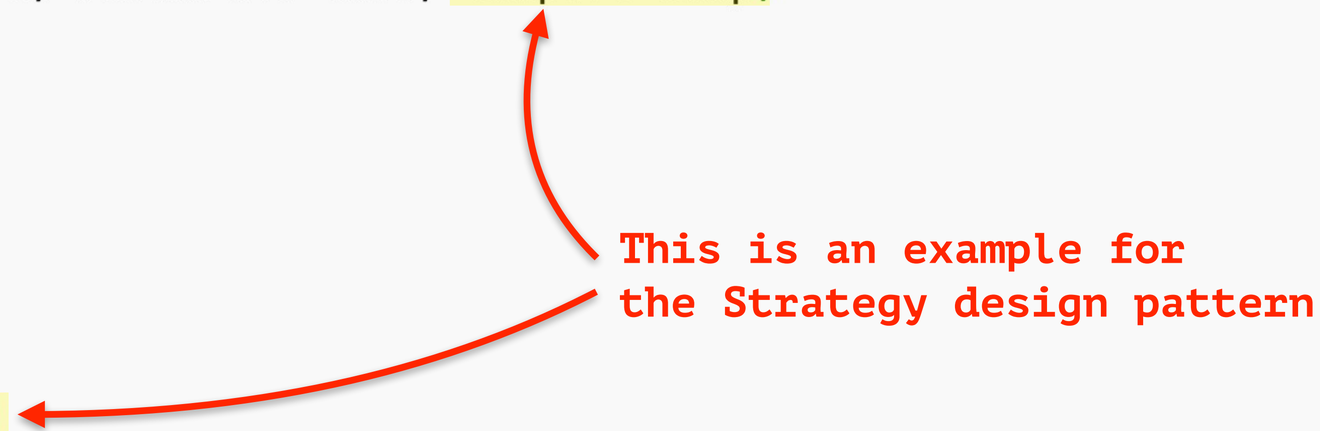
Finds the smallest element in the range [`first` , `last`).

1) Elements are compared using operator<.

```
    return smallest;
}
```

min_element (3)

```
template<class ForwardIt, class Compare>
ForwardIt min_element(ForwardIt first, ForwardIt last, Compare comp)
{
    if (first == last)
        return last;

    ForwardIt smallest = first;
    ++first;

    for (; first != last; ++first)
        if (comp(*first, *smallest))
            smallest = first;

    return smallest;
}
```



Example

Run this code

Introducing std::min_element()

```cpp
int main()
{
   std::vector<Person> table =  /* ... */;

   // ...

   // Find the youngest person
   auto const pos =
      std::min_element( begin(table), end(table),
         []( Person const& lhs, Person const& rhs )
         {
            return lhs.age < rhs.age;
         } );

   if( pos != end(table) ) {
      std::cout << "Youngest person = " << pos->firstname
                << " " << pos->lastname << '\n';
   }

   // ...
```

Easy to read

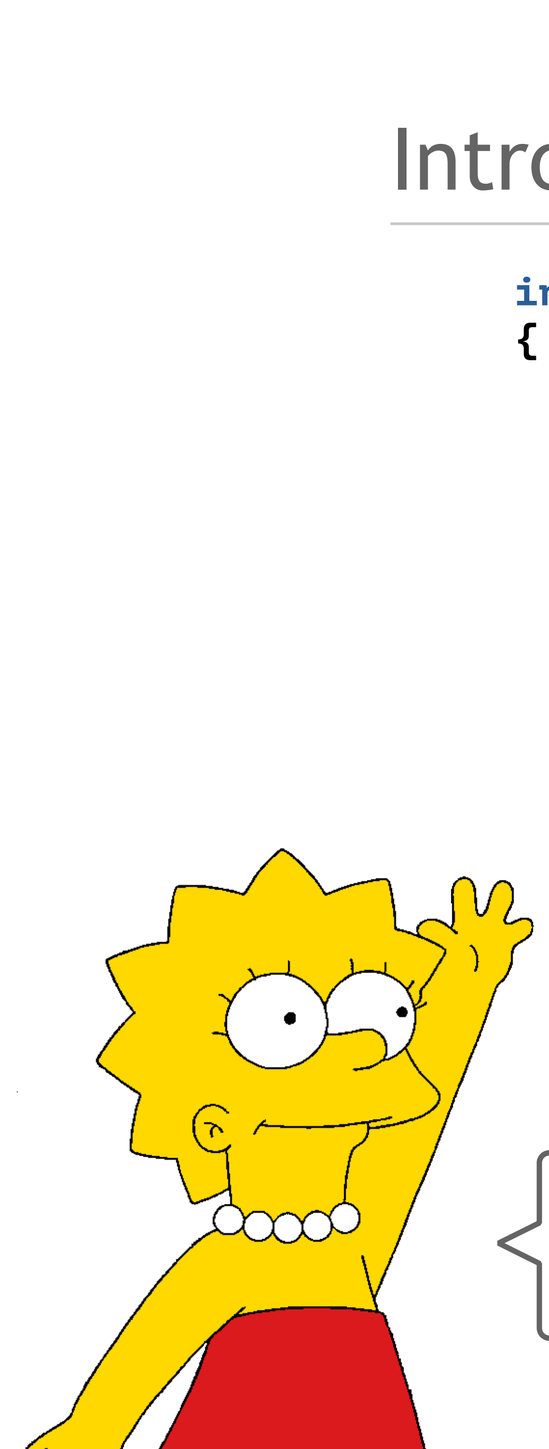
Easy to understand

Less verbose

Easy to get right

Reusable

Introducing std::ranges::min_element()



```cpp
int main()
{
   std::vector<Person> table =  /* ... */;

   // ...

   // Find the youngest person
   auto const pos =
      std::ranges::min_element( table, std::less{}, &Person::age );


   if( pos != end(table) ) {
      std::cout << "Youngest person = " << pos->firstname
                << " " << pos->lastname << '\n';
   }


   // ...
```

You can do even better with std::ranges::min_element().

std::ranges::min_element

Defined in header <algorithm>

Call signature

```
template< std::forward_iterator I, std::sentinel_for<I> S, class Proj = std::identity,
          std::indirect_strict_weak_order<std::projected<I, Proj>> Comp = ranges::less >
constexpr I
    min_element( I first, S last, Comp comp = {}, Proj proj = {} );
```
(1) (since C++20)

```
template< ranges::forward_range R, class Proj = std::identity,
          std::indirect_strict_weak_order<
              std::projected<ranges::iterator_t<R>, Proj>> Comp = ranges::less >
constexpr ranges::borrowed_iterator_t<R>
    min_element( R&& r, Comp comp = {}, Proj proj = {} );
```
(2) (since C++20)

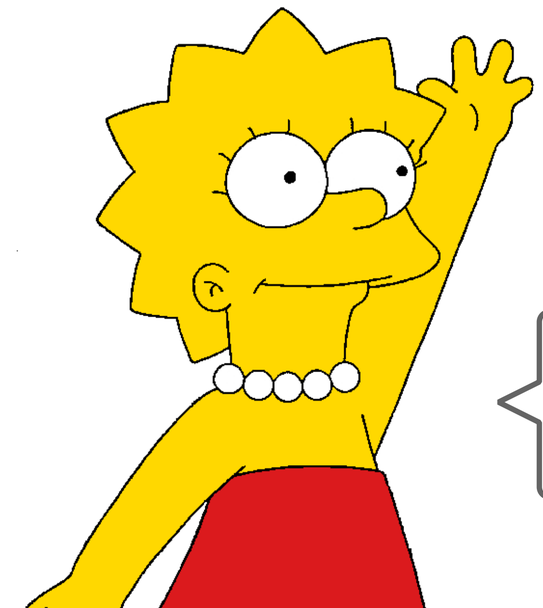
1) Finds the smallest element in the range [`first` , `last`).
2) Same as (1), but uses `r` as the source range, as if using `ranges::begin(r)` as `first` and `ranges::end(r)` as `last` .

The function-like entities described on this page are *niebloids*, that is:

- Explicit template argument lists cannot be specified when calling any of them.
- None of them are visible to argument-dependent lookup.

Introducing std::ranges::min_element()

```
int main()
{
   std::vector<Person> table =  /* ... */;

   // ...

   // Find the youngest person
   auto const pos =
      std::ranges::min_element( table, std::less{}, &Person::age );


   if( pos != end(table) ) {
      std::cout << "Youngest person = " << pos->firstname
                << " " << pos->lastname << '\n';
   }


   // ...
```

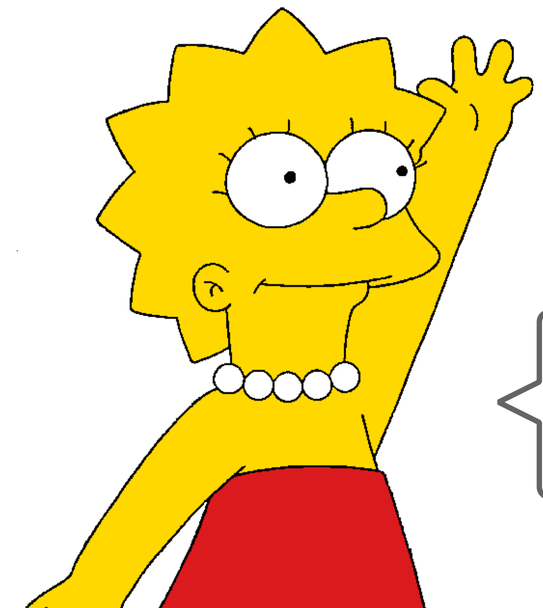


You can even shortcut the second argument to **{}**.

Introducing std::ranges::min_element()

```
int main()
{
    std::vector<Person> table =  /* ... */;

    // ...

    // Find the youngest person
    auto const pos =
        std::ranges::min_element( table, {}, &Person::age );


    if( pos != end(table) ) {
        std::cout << "Youngest person = " << pos->firstname
                  << " " << pos->lastname << '\n';
    }


    // ...
```

The comment can go, the code is expressive enough to be self-documenting



You can even shortcut the second argument to **{}**.

Introducing std::ranges::min_element()

```cpp
int main()
{
    std::vector<Person> table =  /* ... */;

    // ...


    auto const pos =
        std::ranges::min_element( table, {}, &Person::age );


    if( pos != end(table) ) {
        std::cout << "Youngest person = " << pos->firstname
                  << " " << pos->lastname << '\n';
    }


    // ...
```

The Expert’s Advice

”If you want to improve code quality in your organization, I would say, take all your coding guidelines and replace them with the one goal. … No Raw Loops. This will make the biggest change in code quality within your organization.”

(Sean Parent, C++ Seasoning, Going Native 2013)

Fun with the Simpsons

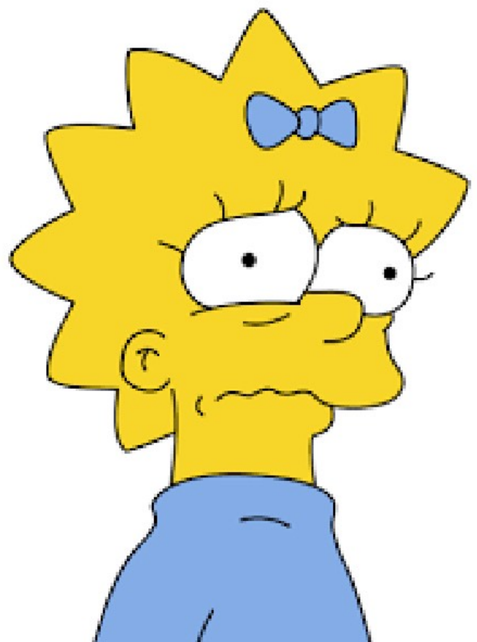
```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector

    // ...
```

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    while( first != last && isChild( *first ) ) ++first;


    if( first != last ) {
       for( auto pos=std::next(first); pos!=last; ++pos ) {
          if( isChild(*pos) ) {
             auto tmp{ *pos };
             *pos = *first;
             *first = tmp;
             ++first;
          }
       }
    }

    // ...
```

Works, right?

Again, Maggie Disapproves the Solution

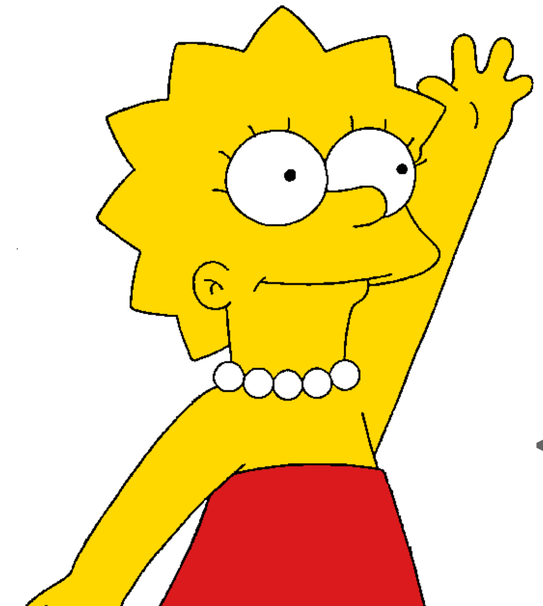


```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    while( first != last && isChild( *first ) ) ++first;


    if( first != last ) {
        for( auto pos=std::next(first); pos!=last; ++pos ) {
            if( isChild(*pos) ) {
                auto tmp{ *pos };
                *pos = *first;
                *first = tmp;
                ++first;
            }
        }
    }

    // ...
```

Again, this comment is a clear indication that something is wrong

Introduction of std::iter_swap()

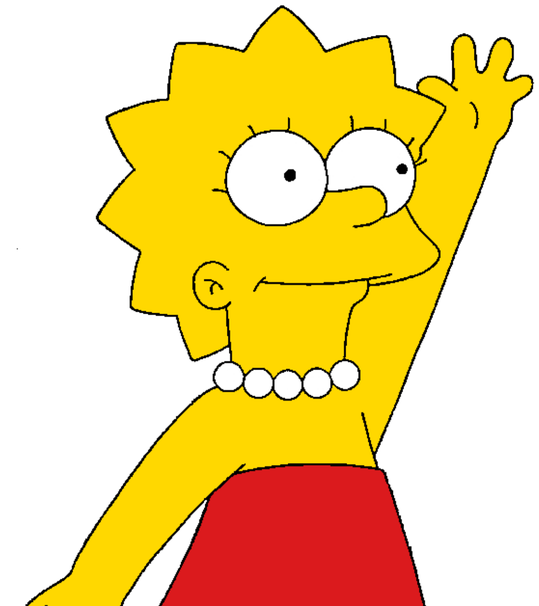
```
int main()
{
   std::vector<Person> table = /* ... */;

   // Put all the children to the beginning of the vector
   auto first = begin(table);
   auto last  = end(table);

   while( first != last && isChild( *first ) ) ++first;


   if( first != last ) {
      for( auto pos=std::next(first); pos!=last; ++pos ) {
         if( isChild(*pos) ) {
            auto tmp{ *pos };
            *pos = *first;
            *first = tmp;
            ++first;
         }
```



This is a common pattern, the pattern of an algorithm. This is a …

Introduction of std::iter_swap()

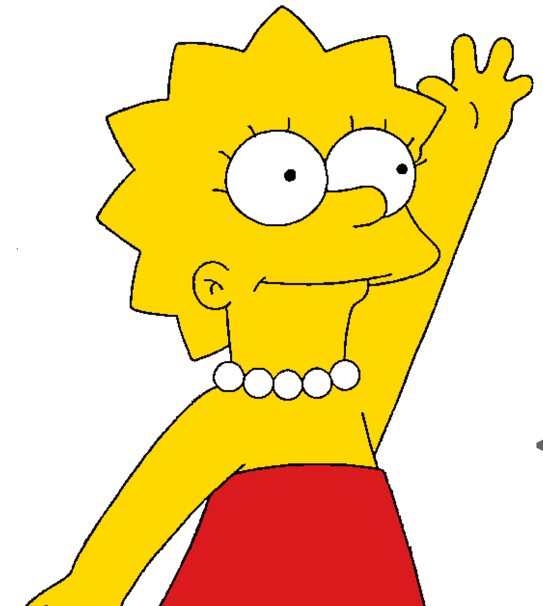
```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    while( first != last && isChild( *first ) ) ++first;


    if( first != last ) {
        for( auto pos=std::next(first); pos!=last; ++pos ) {
            if( isChild(*pos) ) {
                std::iter_swap(pos, first);
                ++first;
            }
        }
    }
}
```



... std::iter_swap()!

Introduction of std::find_if_not()

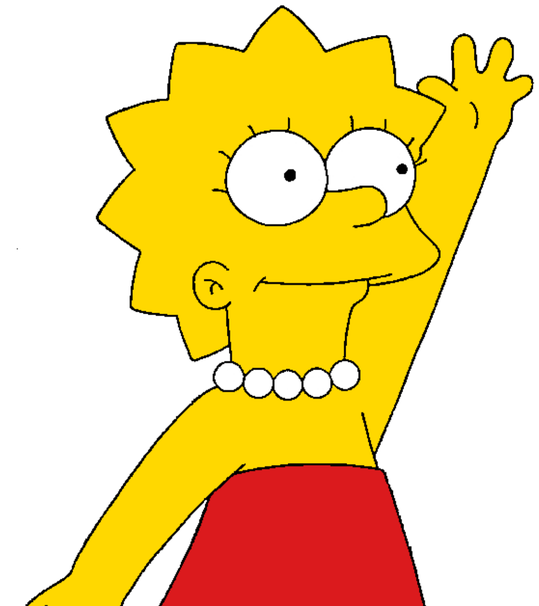
```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    while( first != last && isChild( *first ) ) ++first;


    if( first != last ) {
        for( auto pos=std::next(first); pos!=last; ++pos ) {
            if( isChild(*pos) ) {
                std::iter_swap(pos, first);
                ++first;
            }
        }
    }
}
```



Oh, that’s another common pattern. This time it is a …

Introduction of std::find_if_not()

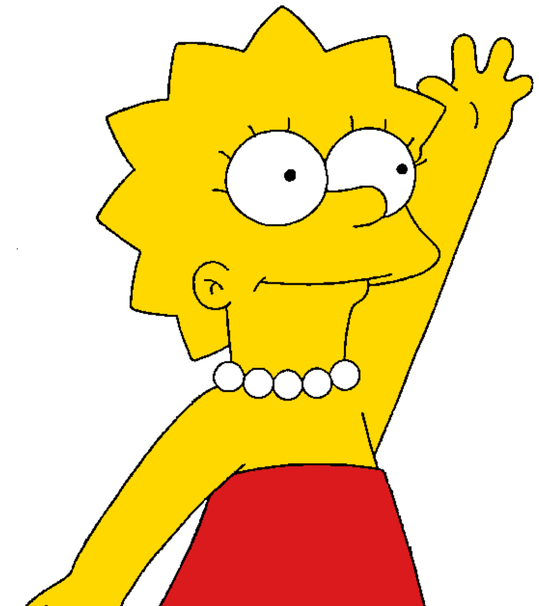
```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    first = std::find_if_not( first, last, isChild );


    if( first != last ) {
       for( auto pos=std::next(first); pos!=last; ++pos ) {
          if( isChild(*pos) ) {
             std::iter_swap(pos, first);
             ++first;
          }
       }
    }
}
```



... std::find_if_not()!

Introduction of std::find_if_not()

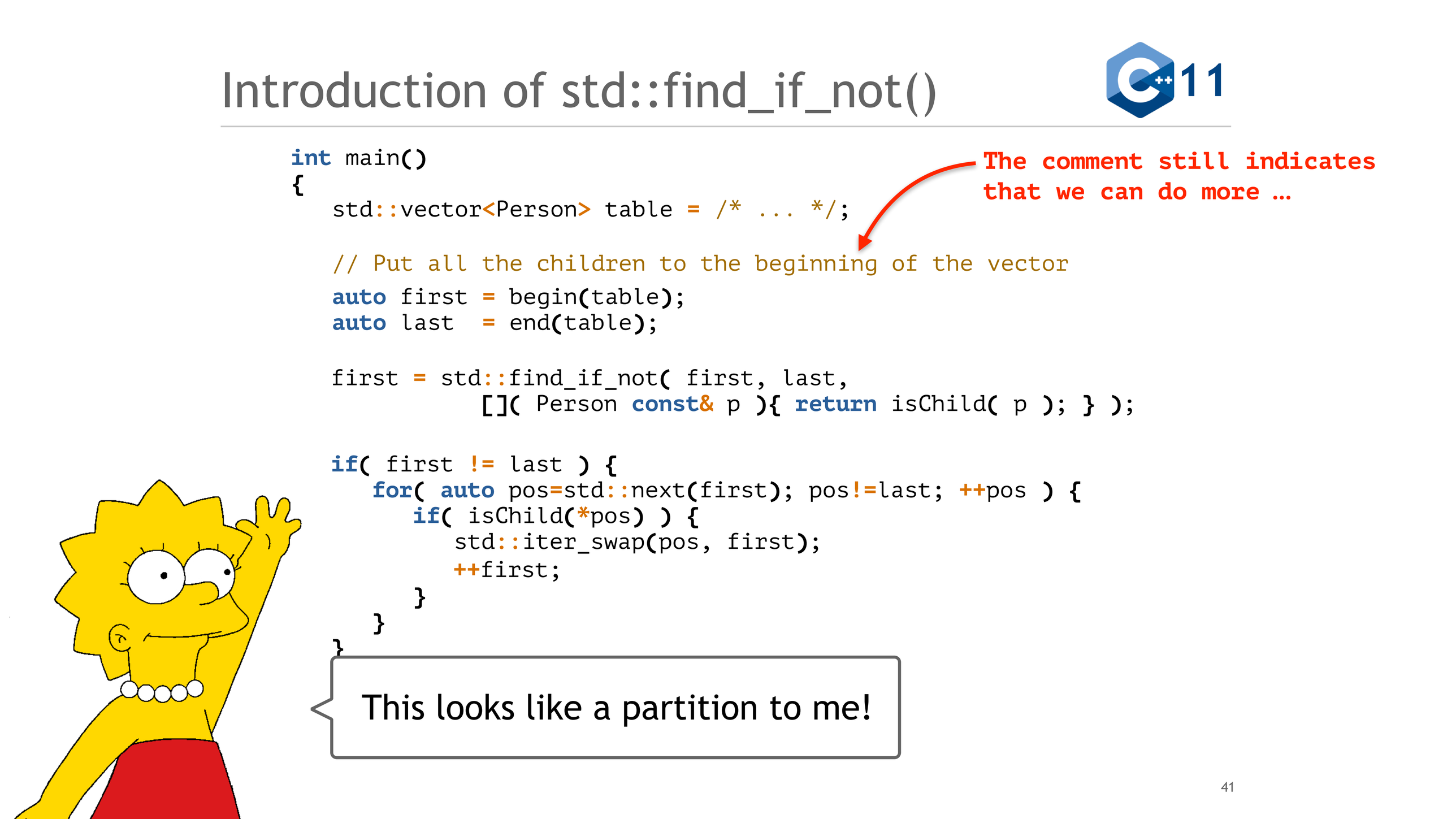
```
int main()
{
   std::vector<Person> table = /* ... */;

   // Put all the children to the beginning of the vector
   auto first = begin(table);
   auto last  = end(table);

   first = std::find_if_not( first, last,
               []( Person const& p ){ return isChild( p ); } );

   if( first != last ) {
      for( auto pos=std::next(first); pos!=last; ++pos ) {
         if( isChild(*pos) ) {
            std::iter_swap(pos, first);
            ++first;
         }
      }
   }
}
```



Prefer to pass a function object, though.

(see Core Guideline T.40 for the reasons why).

Introduction of std::find_if_not()




```
int main()
{
    std::vector<Person> table = /* ... */;

    // Put all the children to the beginning of the vector
    auto first = begin(table);
    auto last  = end(table);

    first = std::find_if_not( first, last,
                []( Person const& p ){ return isChild( p ); } );

    if( first != last ) {
       for( auto pos=std::next(first); pos!=last; ++pos ) {
          if( isChild(*pos) ) {
             std::iter_swap(pos, first);
             ++first;
          }
       }
    }
}
```

The comment still indicates that we can do more …

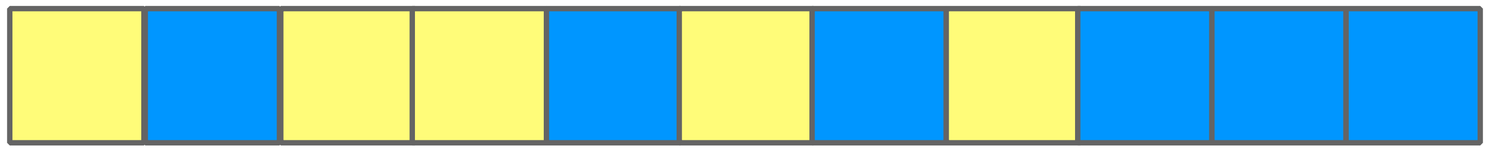
This looks like a partition to me!

Introduction of std::partition()


```
int main()
{
   std::vector<Person> table = /* ... */;


   auto first = begin(table);
   auto last  = end(table);

   std::partition( first, last, []( Person const& p )
   {
      return isChild( p );
   } );

   // ...
```

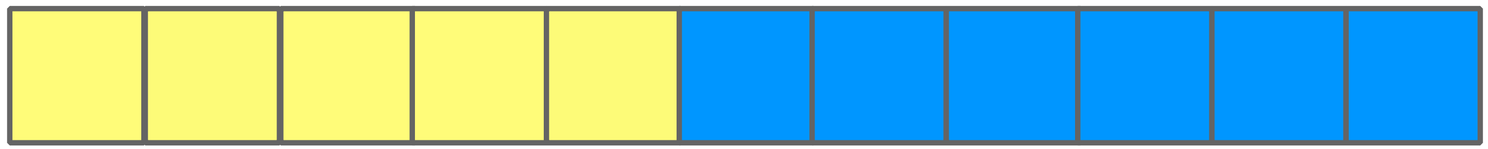
Introduction of std::partition()

```
int main()
{
    std::vector<Person> table = /* ... */;


    auto first = begin(table);
    auto last  = end(table);

    std::partition( first, last, [](  Person const& p )
    {
       return isChild( p );
    } );

    // ...
```



Introduction of std::partition()

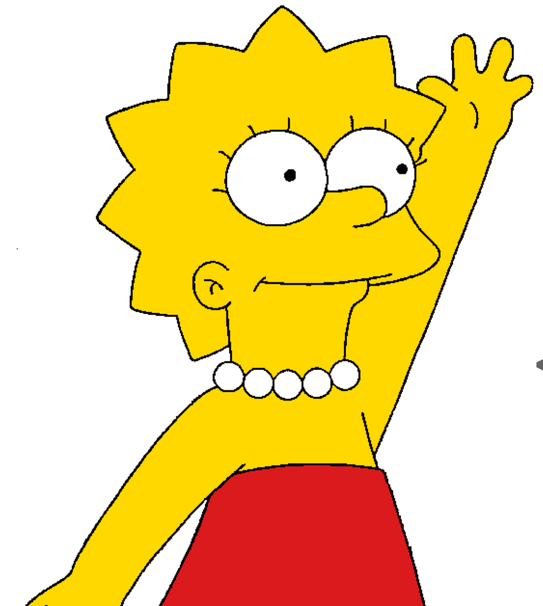
```
int main()
{
    std::vector<Person> table = /* ... */;


    auto first = begin(table);
    auto last  = end(table);

    std::partition( first, last, [](  Person const& p )
    {
       return isChild( p );
    } );

    // ...
```



Algorithms use Algorithms

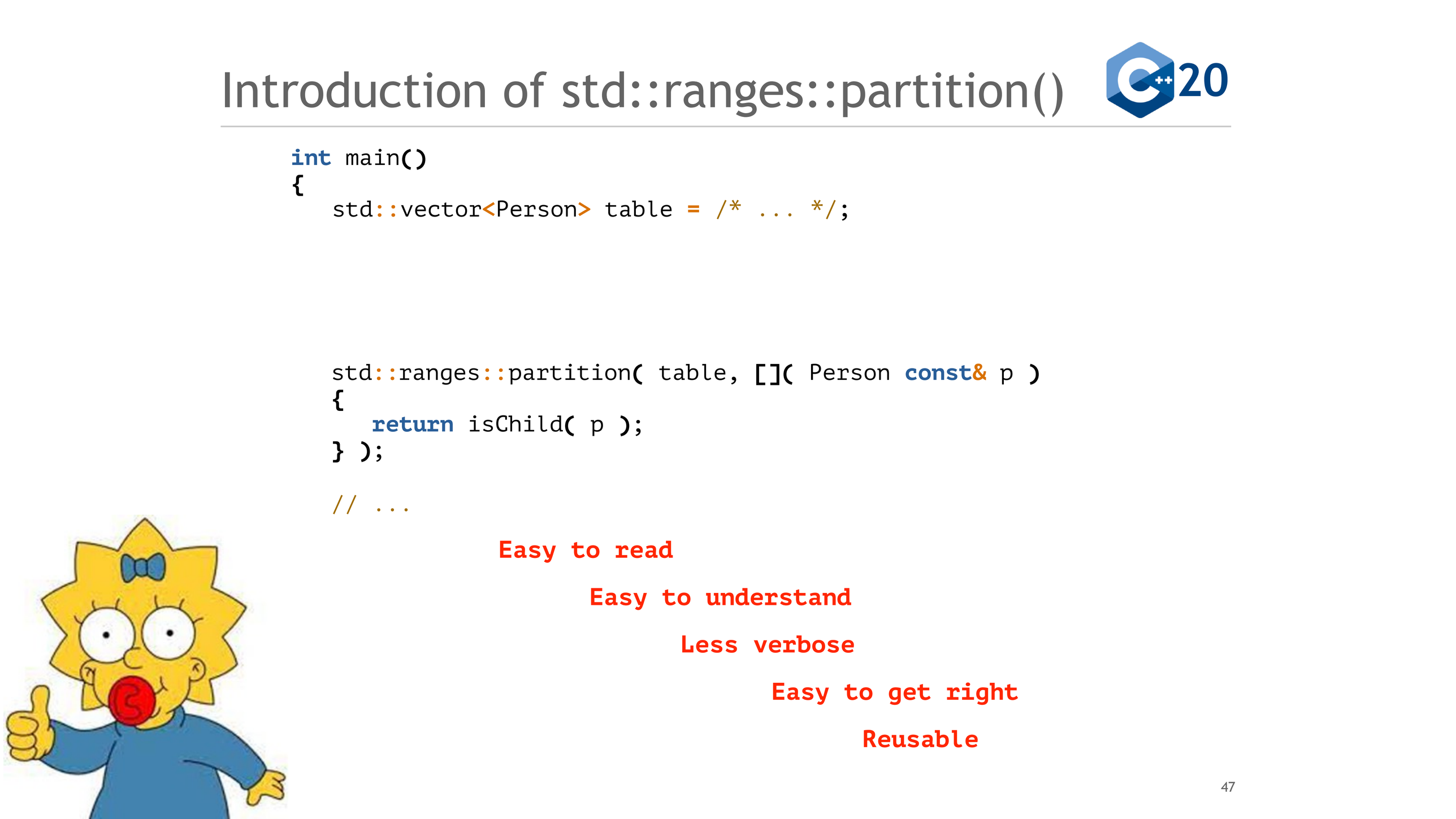
Introduction of std::partition()

```
int main()
{
    std::vector<Person> table = /* ... */;


    auto first = begin(table);
    auto last  = end(table);

    std::partition( first, last, [](  Person const& p )
    {
       return isChild( p );
    } );

    // ...
```



Again we can do better by using the C++20 ranges form of partition.

Introduction of std::ranges::partition()




```
int main()
{
   std::vector<Person> table = /* ... */;
```

```
   std::ranges::partition( table, []( Person const& p )
   {
      return isChild( p );
   } );

   // ...
```

Easy to read

Easy to understand

Less verbose

Easy to get right

Reusable

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;




    std::ranges::partition( table, []( Person const& p )
    {
       return isSeniorCitizen( p );
    } );

    // ...
```

That’s perfect reuse!

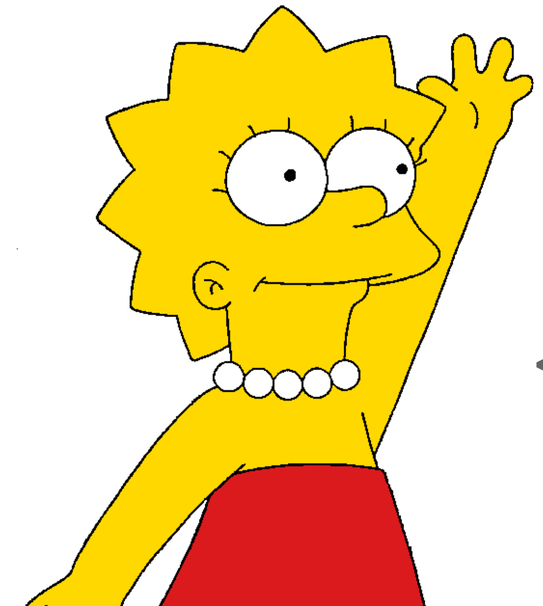
Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    std::ranges::partition( table, []( Person const& p )
    {
        return isSeniorCitizen( p );
    } );

    // ...
```

What if I wanted to keep the previous order of the senior citizens?

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    std::ranges::partition( table, []( Person const& p )
    {
        return isSeniorCitizen( p );
    } );

    // ...
```



What if I wanted to keep the previous order of the senior citizens?

Don’t reinvent, look around! This is a perfect opportunity for stable_partition()!

Partitioning Algorithms

Partitioning operations

Defined in header `<algorithm>`

is_partitioned (C++11)	determines if the range is partitioned by the given predicate (function template)
ranges::is_partitioned (C++20)	determines if the range is partitioned by the given predicate (niebloid)
partition	divides a range of elements into two groups (function template)
ranges::partition (C++20)	divides a range of elements into two groups (niebloid)
partition_copy (C++11)	copies a range dividing the elements into two groups (function template)
ranges::partition_copy (C++20)	copies a range dividing the elements into two groups (niebloid)
stable_partition	divides elements into two groups while preserving their relative order (function template)
ranges::stable_partition (C++20)	divides elements into two groups while preserving their relative order (niebloid)
partition_point (C++11)	locates the partition point of a partitioned range (function template)
ranges::partition_point (C++20)	locates the partition point of a partitioned range (niebloid)

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    std::ranges::stable_partition( table, []( Person const& p )
    {
        return isSeniorCitizen( p );
    } );

    // ...
```

The naming conversion is pretty consistent (although not perfect).

Cool!

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    std::ranges::stable_partition( table, []( Person const& p )
    {
        return isSeniorCitizen( p );
    } );

    // ...
```

What if I now wanted to put the senior citizens last, without changing their order?

Well, that's a rotate!

Fun with the Simpsons

```
int main()
{
   std::vector<Person> table = /* ... */;

   auto const pos =
      std::ranges::stable_partition( table, []( Person const& p )
      {
         return isSeniorCitizen( p );
      } );

   std::rotate( begin(table), pos, end(table) );

   // ...
```

Wow, that's simple and elegant!
What would happen if I pass pos last?

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    auto const pos =
       std::ranges::stable_partition( table, []( Person const& p )
       {
          return isSeniorCitizen( p );
       } );

    std::rotate( begin(table), end(table), pos );

    // ...
```

Well, that would be UB.

Wow, that's simple and elegant!
What would happen if I pass pos last?

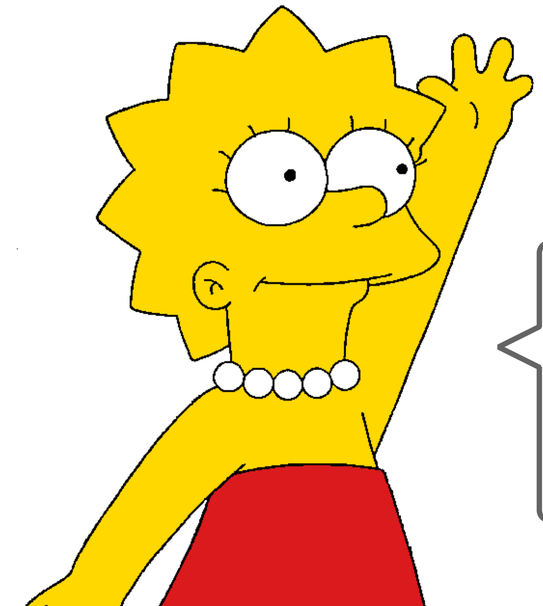
Fun with the Simpsons

```
int main()
{
   std::vector<Person> table = /* ... */;

   auto const pos =
      std::ranges::stable_partition( table, []( Person const& p )
      {
         return isSeniorCitizen( p );
      } );

   std::ranges::rotate( table, pos );

   // ...
```

Therefore prefer the C++20 ranges algorithm.

Fun with the Simpsons

```cpp
int main()
{
    std::vector<Person> table = /* ... */;

    // Sum up the age of all persons
    int total_age = 0;
    for( auto const& person : table ) {
       total_age += person.age;
    }
    std::cout << "Total age = " << total_age << '\n';

    // ...
```



Not entirely bad: it’s short and clear. But you could also use the accumulate() algorithm.

Now I need to sum up the age of all persons.

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                    (1)  (until C++20)
template< class InputIt, class T >
constexpr T accumulate( InputIt first, InputIt last, T init );          (1)  (since C++20)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );                                     (2)  (until C++20)
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );                           (2)  (since C++20)
```

Computes the sum of the given value `init` and the elements in the range [`first` , `last`).

1) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = acc + *i` (until C++20) `acc = std::move(acc) + *i` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

2) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = op(acc, *i)` (until C++20) `acc = op(std::move(acc), *i)` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

Possible implementation

accumulate (1)

```
template<class InputIt, class T>
constexpr // since C++20
T accumulate(InputIt first, InputIt last, T init)
{
    for (; first != last; ++first)
        init = std::move(init) + *first; // std::move since C++20

    return init;
}
```

accumulate (2)

```
template<class InputIt, class T, class BinaryOperation>
constexpr // since C++20
T accumulate(InputIt first, InputIt last, T init, BinaryOperation op)
{
    for (; first != last; ++first)
        init = op(std::move(init), *first); // std::move since C++20

    return init;
}
```

Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Sum up the age of all persons
    auto const total_age =
       std::accumulate( std::begin(table), std::end(table), 0,
          []( int age, Person const& p )
          {
             return age + p.age;
          } );
    std::cout << "Total age = " << total_age << '\n';

    // ...
```

Now why exactly is this better ???

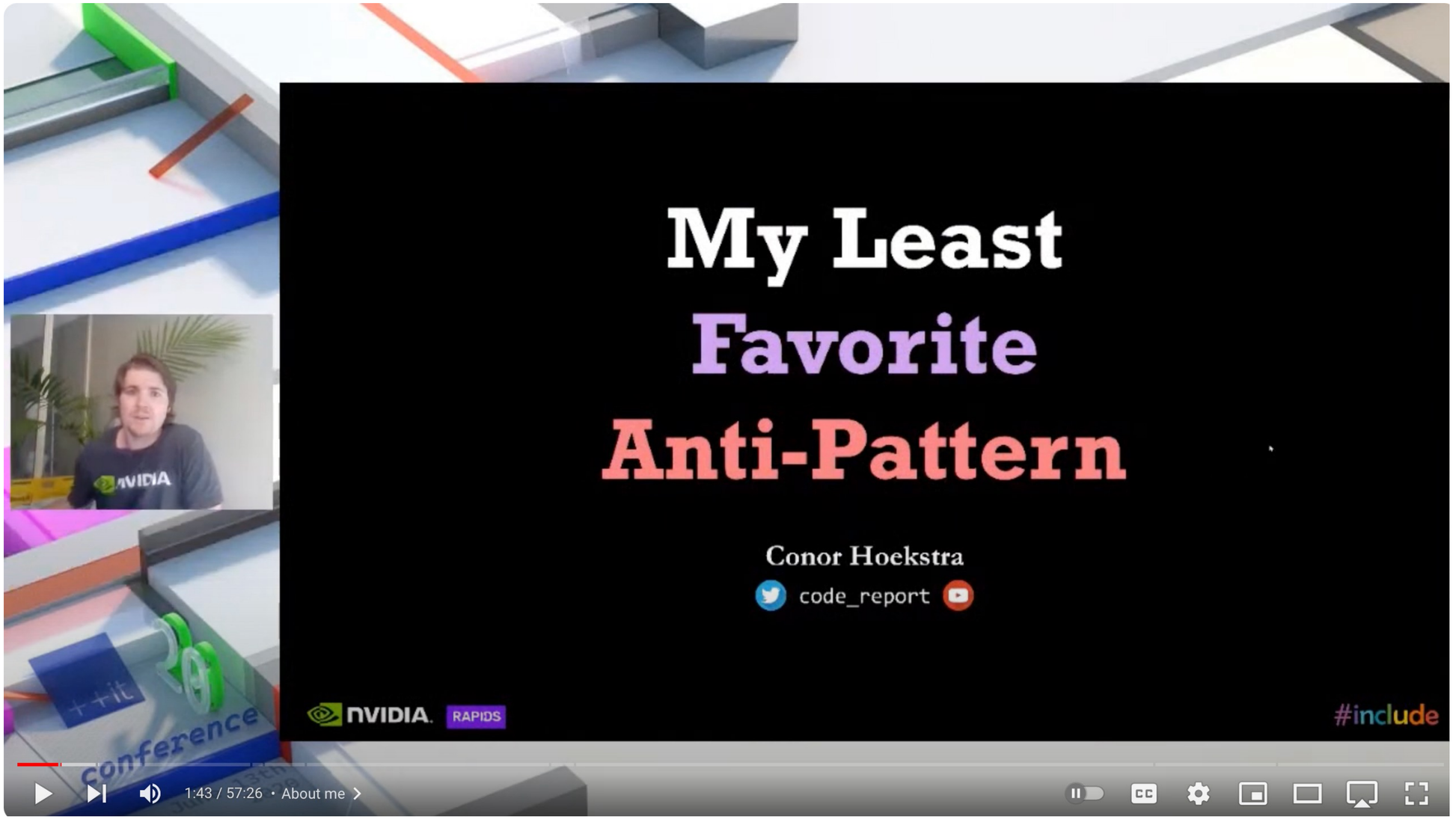
Fun with the Simpsons

```cpp
int main()
{
    std::vector<Person> table = /* ... */;

    // Sum up the age of all persons
    auto const total_age =
        std::accumulate( std::begin(table), std::end(table), 0,
            []( int age, Person const& p )
            {
                return age + p.age;
            } );
    std::cout << "Total age = " << total_age << '\n';

    // ...
```

Look at the const! It avoids the ITM anti-pattern.

Now why exactly is this better ???

Conor Hoekstra on “Initialize Then Modify”



Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Sum up the age of all persons
    auto const total_age =
       std::accumulate( std::begin(table), std::end(table), 0,
          []( int age, Person const& p )
          {
             return age + p.age;
          } );
    std::cout << "Total age = " << total_age << '\n';

    // ...
```

Prefer to keep things immutable. This helps!

What about performance?
Isn't the for loop faster?

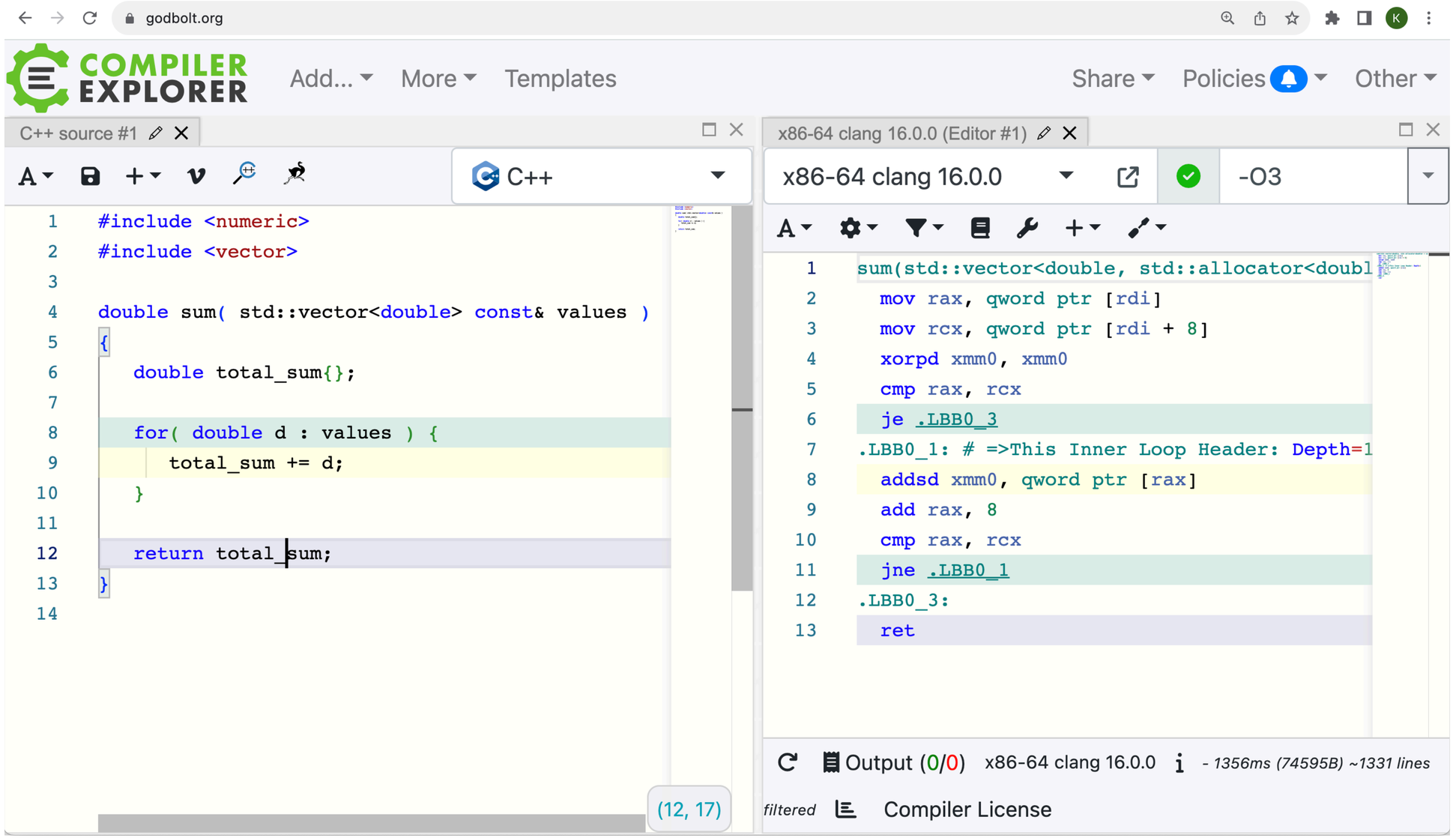
Fun with the Simpsons

```
int main()
{
    std::vector<Person> table = /* ... */;

    // Sum up the age of all persons
    auto const total_age =
       std::accumulate( std::begin(table), std::end(table), 0,
          []( int age, Person const& p )
          {
             return age + p.age;
          } );
    std::cout << "Total age = " << total_age << '\n';

    // ...
```

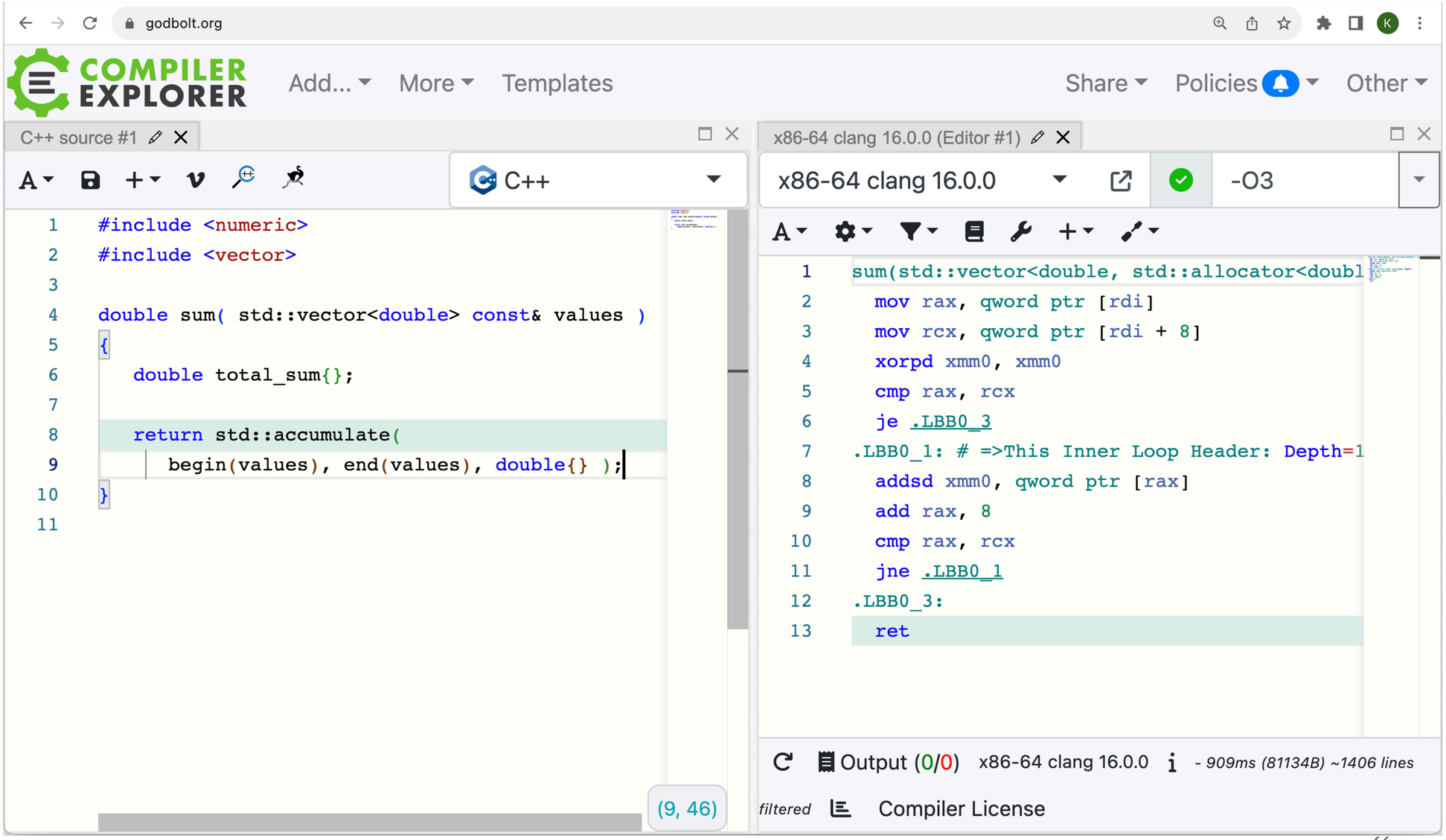
No, it’s not. Let me show you…

What about performance?
Isn’t the for loop faster?

The Performance of Algorithms



The Performance of Algorithms

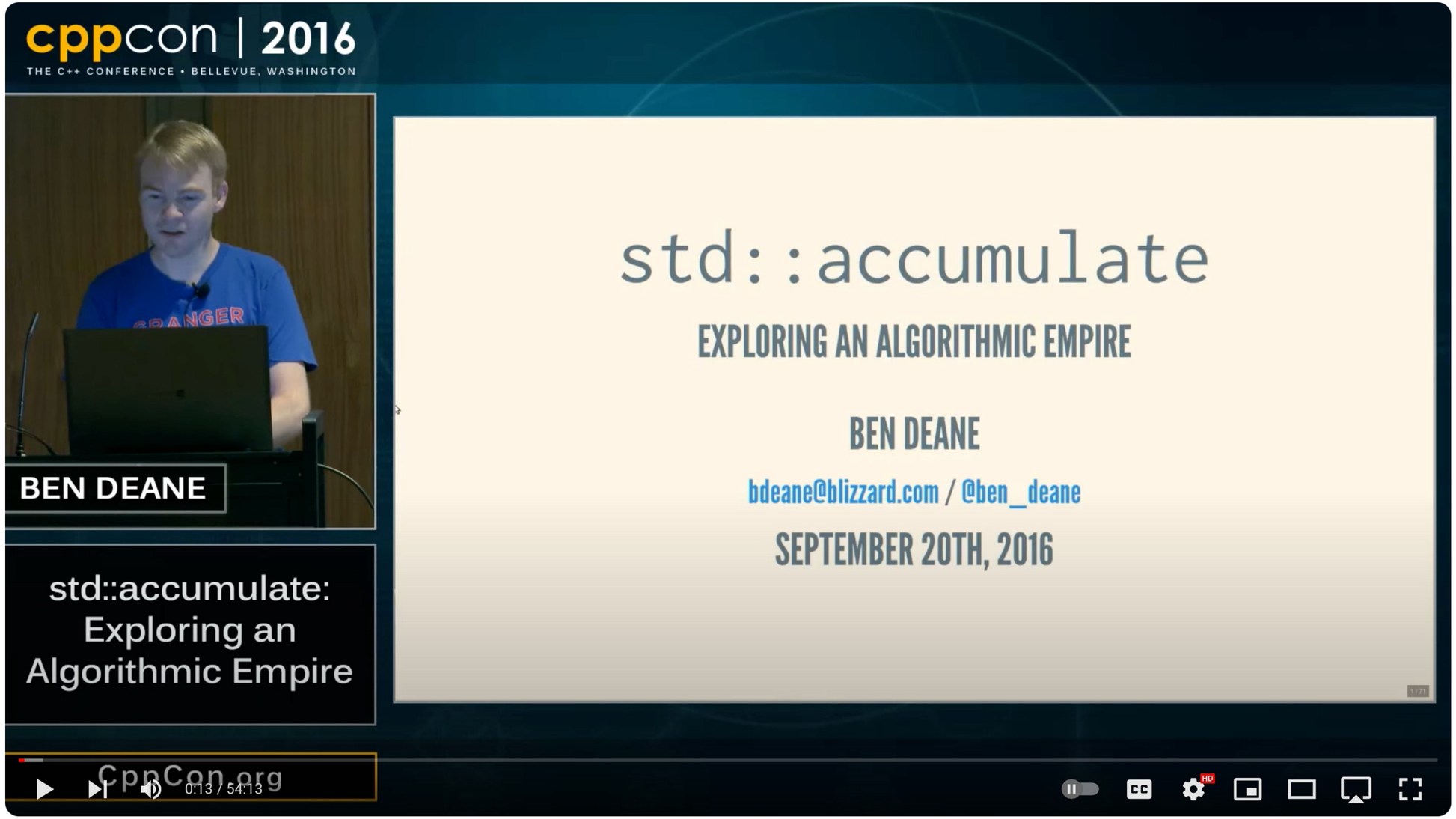


Fun with the Simpsons

```
int main()
{
   std::vector<Person> table = /* ... */;

   // Sum up the age of all persons
   auto const total_age =
      std::accumulate( std::begin(table), std::end(table), 0,
         []( int age, Person const& p )
         {
            return age + p.age;
         } );
   std::cout << "Total age = " << total_age << '\n';

   // ...
```

Wow, they are fast AND elegant! I'm deeply impressed.
I'm still wondering why accumulate is so complex…

Ben Deane on std::accumulate()



Fun with the Simpsons

```
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::accumulate( std::begin(values), std::end(values), 0 );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

Cool! And it is perfectly reusable again. Let's use this on a vector of floating-point values …

Fun with the Simpsons

```
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::accumulate( std::begin(values), std::end(values), 0 );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

Cool! And it is perfectly reusable again. Let's use this on a vector of floating-point values …

But please take care of the initial value!

std::accumulate

Defined in header <numeric>

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );                    (1)  (until C++20)
template< class InputIt, class T >
constexpr T accumulate( InputIt first, InputIt last, T init );          (1)  (since C++20)

template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );                                     (2)  (until C++20)
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
                        BinaryOperation op );                           (2)  (since C++20)
```

Computes the sum of the given value `init` and the elements in the range [`first` , `last`).

1) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = acc + *i` (until C++20) `acc = std::move(acc) + *i` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

2) Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with `acc = op(acc, *i)` (until C++20) `acc = op(std::move(acc), *i)` (since C++20) for every iterator `i` in the range [`first` , `last`) in order.

Fun with the Simpsons

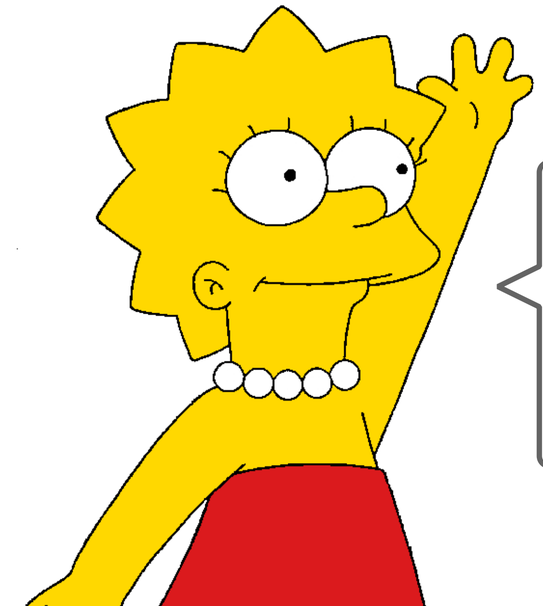

```cpp
int main()
{
   std::vector<double> values =  /* ... */;

   // Sum up the age of all persons
   auto const total_sum =
      std::accumulate( std::begin(values), std::end(values), 0 );

   std::cout << "Total sum = " << total_sum << '\n';

   // ...
```

Fun with the Simpsons


```cpp
int main()
{
   std::vector<double> values =  /* ... */;

   // Sum up the age of all persons
   auto const total_sum =
      std::accumulate( std::begin(values), std::end(values), 0.0 );

   std::cout << "Total sum = " << total_sum << '\n';

   // ...
```

Fun with the Simpsons

```cpp
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::accumulate( std::begin(values), std::end(values), double{} );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```



Explicitly mentioning the type might be even better

(and might protect against those that still work with copy and paste)!

Fun with the Simpsons

```cpp
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::accumulate( std::begin(values), std::end(values), double{} );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

What if I wanted to speed this up with many threads?

Fun with the Simpsons

```cpp
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::accumulate( std::begin(values), std::end(values), double{} );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

What if I wanted to speed this up with many threads?

Then you would use C++17 std::reduce() instead!

std::reduce

Defined in header <numeric>

```
template< class InputIt >
typename std::iterator_traits<InputIt>::value_type
    reduce( InputIt first, InputIt last );                                  (1)  (since C++17) (until C++20)

template< class InputIt >
constexpr typename std::iterator_traits<InputIt>::value_type
    reduce( InputIt first, InputIt last );                                       (since C++20)

template< class ExecutionPolicy, class ForwardIt >
typename std::iterator_traits<ForwardIt>::value_type
    reduce( ExecutionPolicy&& policy,
            ForwardIt first, ForwardIt last );                              (2)  (since C++17)

template< class InputIt, class T >
T reduce( InputIt first, InputIt last, T init );                            (3)  (since C++17) (until C++20)

template< class InputIt, class T >
constexpr T reduce( InputIt first, InputIt last, T init );                   (since C++20)

template< class ExecutionPolicy, class ForwardIt, class T >
T reduce( ExecutionPolicy&& policy,
          ForwardIt first, ForwardIt last, T init );                        (4)  (since C++17)

template< class InputIt, class T, class BinaryOp >                               (since C++17)
```

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several execution policies, and the library provides corresponding execution policy types and objects. Users may select an execution policy statically by invoking a parallel algorithm with an execution policy object of the corresponding type.

Standard library implementations (but not the users) may define additional execution policies as an extension. The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type is implementation-defined.

Parallel version of algorithms (except for `std::for_each` and `std::for_each_n`) are allowed to make arbitrary copies of elements from ranges, as long as both `std::is_trivially_copy_constructible_v<T>` and `std::is_trivially_destructible_v<T>` are `true`, where T is the type of elements.

Defined in header `<execution>`
Defined in namespace `std::execution`

sequenced_policy (C++17) **parallel_policy** (C++17) **parallel_unsequenced_policy** (C++17) **unsequenced_policy** (C++20)	execution policy types (class)	(since C++17)
seq (C++17) **par** (C++17) **par_unseq** (C++17) **unseq** (C++20)	global execution policy objects (constant)	

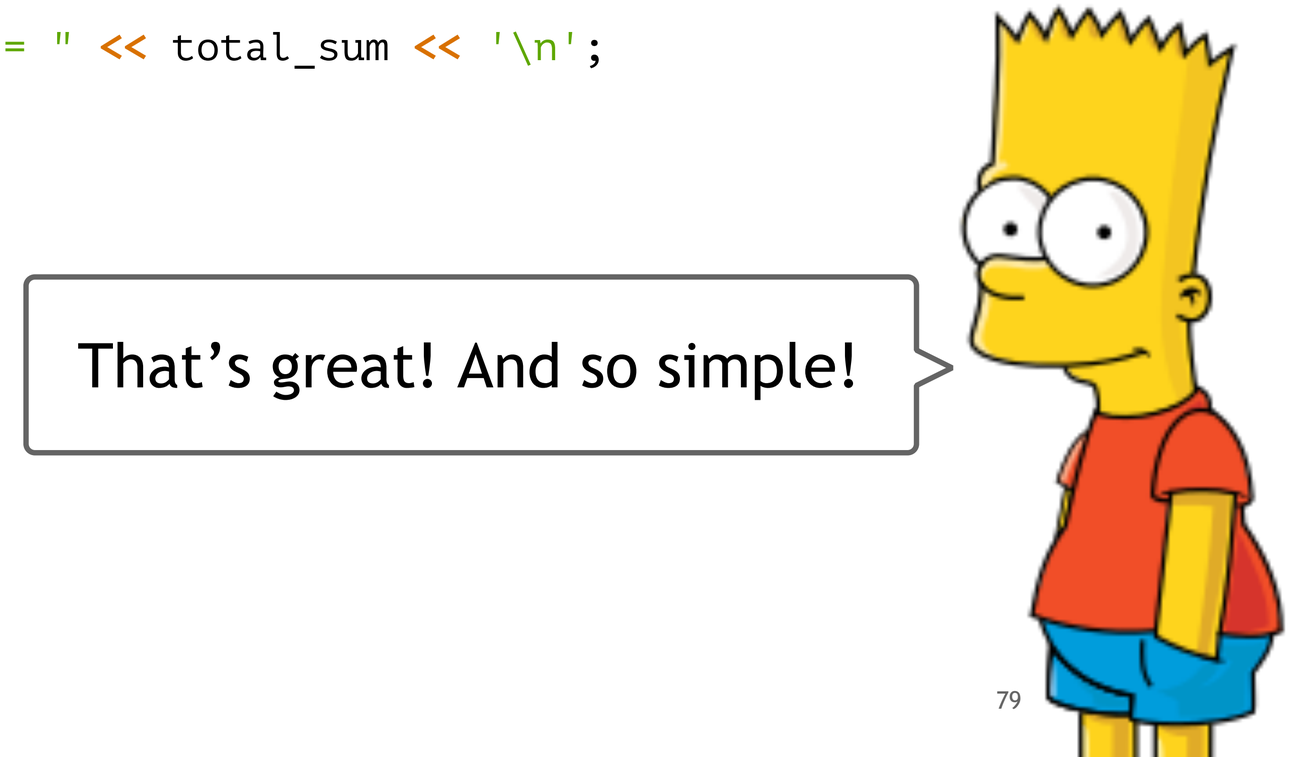
Defined in namespace `std`

is_execution_policy (C++17)	test whether a class represents an execution policy (class template)

Feature-test macro	Value	Std	Feature
`__cpp_lib_parallel_algorithm`	`201603L`	(C++17)	Parallel algorithms

Fun with the Simpsons

```cpp
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::reduce( std::execution::par,
           std::begin(values), std::end(values), double{} );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```



Fun with the Simpsons

```
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::reduce( std::execution::par,
          std::begin(values), std::end(values), double{} );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

That’s great! And so simple!

You could even drop the initial value here!

Fun with the Simpsons

```
int main()
{
    std::vector<double> values =  /* ... */;

    // Sum up the age of all persons
    auto const total_sum =
       std::reduce( std::execution::par,
           std::begin(values), std::end(values) );

    std::cout << "Total sum = " << total_sum << '\n';

    // ...
```

Wow, that’s so cool! I love this!

You could even drop the initial value here!

The Expert’s Advice

”If you want to improve code quality in your organization, I would say, take all your coding guidelines and replace them with the one goal. … No Raw Loops. This will make the biggest change in code quality within your organization.”

(Sean Parent, C++ Seasoning, Going Native 2013)

I completely agree now! But does this mean that I should not use for or while loops anymore?

Fun with the Simpsons

```
std::ostream& operator<<( std::ostream& os, Person const& person )
{
   /*...*/ return os;
}

int main()
{
   std::vector<Person> table = /* ... */;

   std::copy( std::begin(table), std::end(table),
      std::ostream_iterator<Person>( std::cout, "\n" ) );

   // ...
```

Let’s assume I just want to print all the persons.
Do I have to use an algorithm, e.g. std::copy() …

Fun with the Simpsons

```
std::ostream& operator<<( std::ostream& os, Person const& person )
{
   /*...*/ return os;
}

int main()
{
   std::vector<Person> table = /* ... */;

   std::for_each( std::begin(table), std::end(table),
      []( Person const& p )
      {
         std::cout << p << '\n';
      } );

   // ...
```

… or std::for_each()? Isn’t that overly complex?

Fun with the Simpsons

```
std::ostream& operator<<( std::ostream& os, Person const& person )
{
   /*...*/ return os;
}

int main()
{
   std::vector<Person> table = /* ... */;

   for( auto const& p : table ) {
      std::cout << p << '\n';
   }

   // ...
```

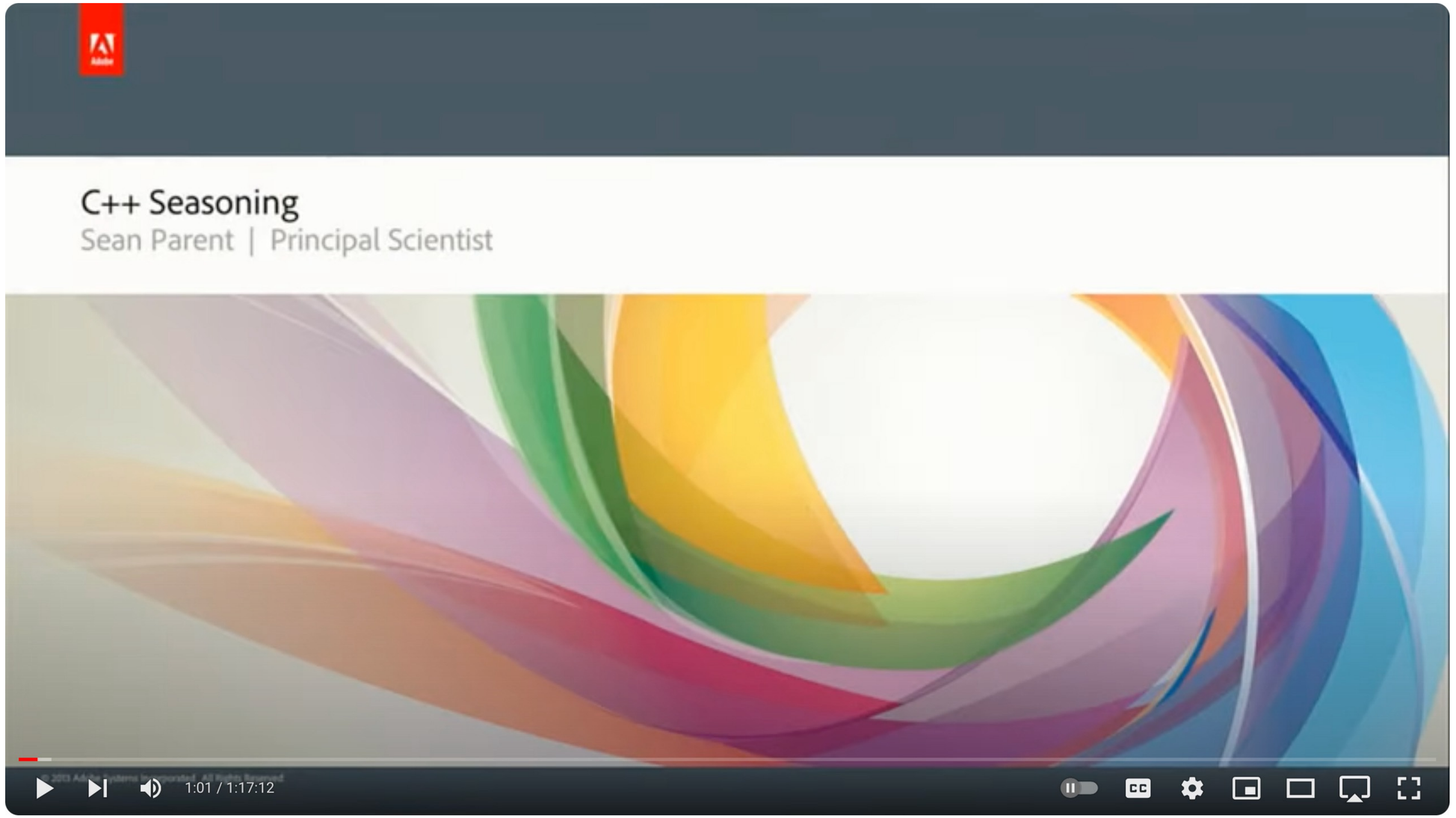
It definitely helps to know your algorithms, but it is not forbidden to use for loops …

Fun with the Simpsons

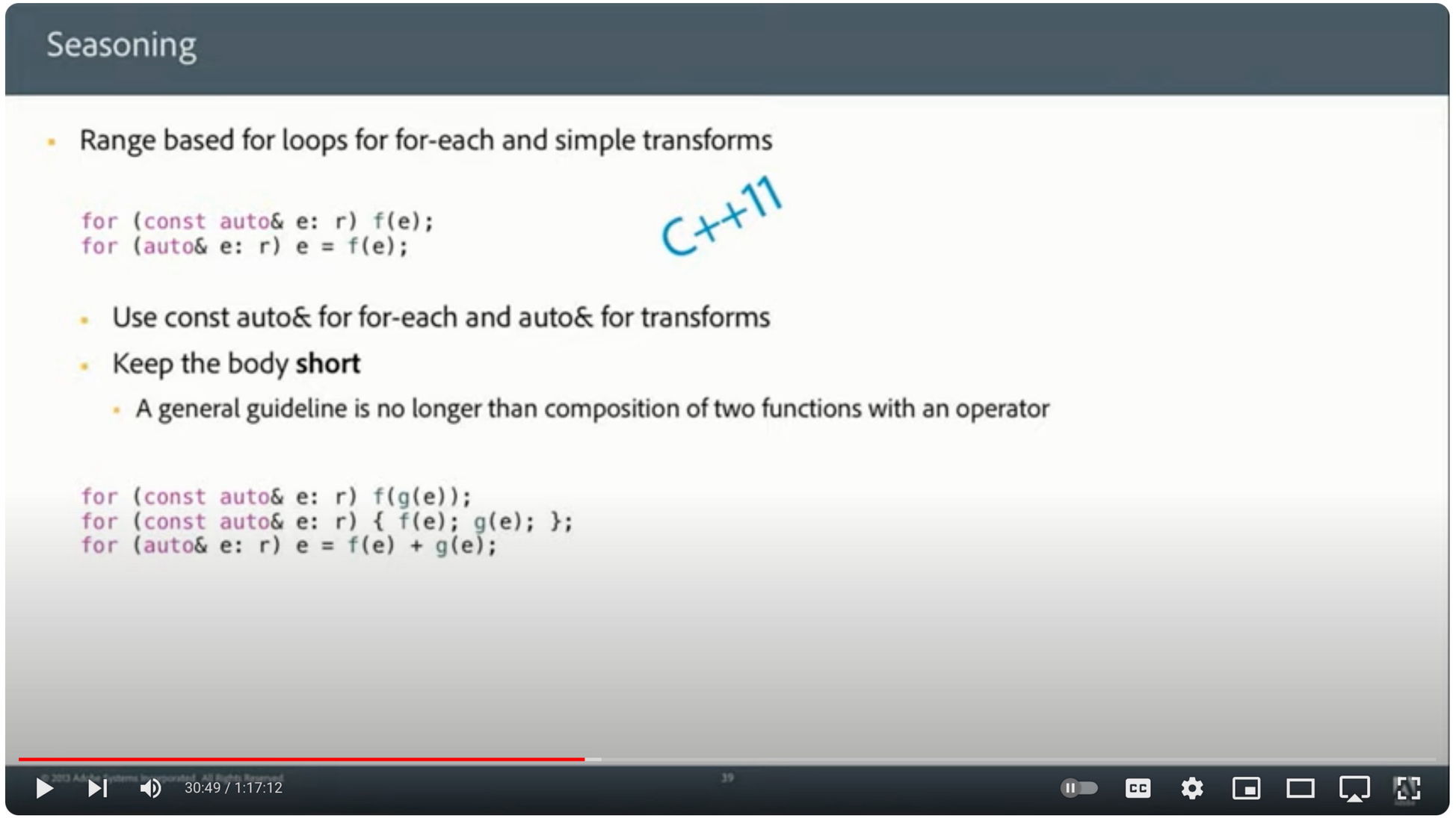
```
std::ostream& operator<<( std::ostream& os, Person const& person )
{
   /*...*/ return os;
}

int main()
{
   std::vector<Person> table = /* ... */;

   for( auto const& p : table ) {
      std::cout << p << '\n';
   }

   // ...
```

… if they are simple and short!

A Definition of Raw Loops



A Definition of Raw Loops



The Expert’s Interpretation of Raw Loops

”9 times out of 10, a for-loop should either be the only code in a function, or the only code in the loop should be a function (or both).”

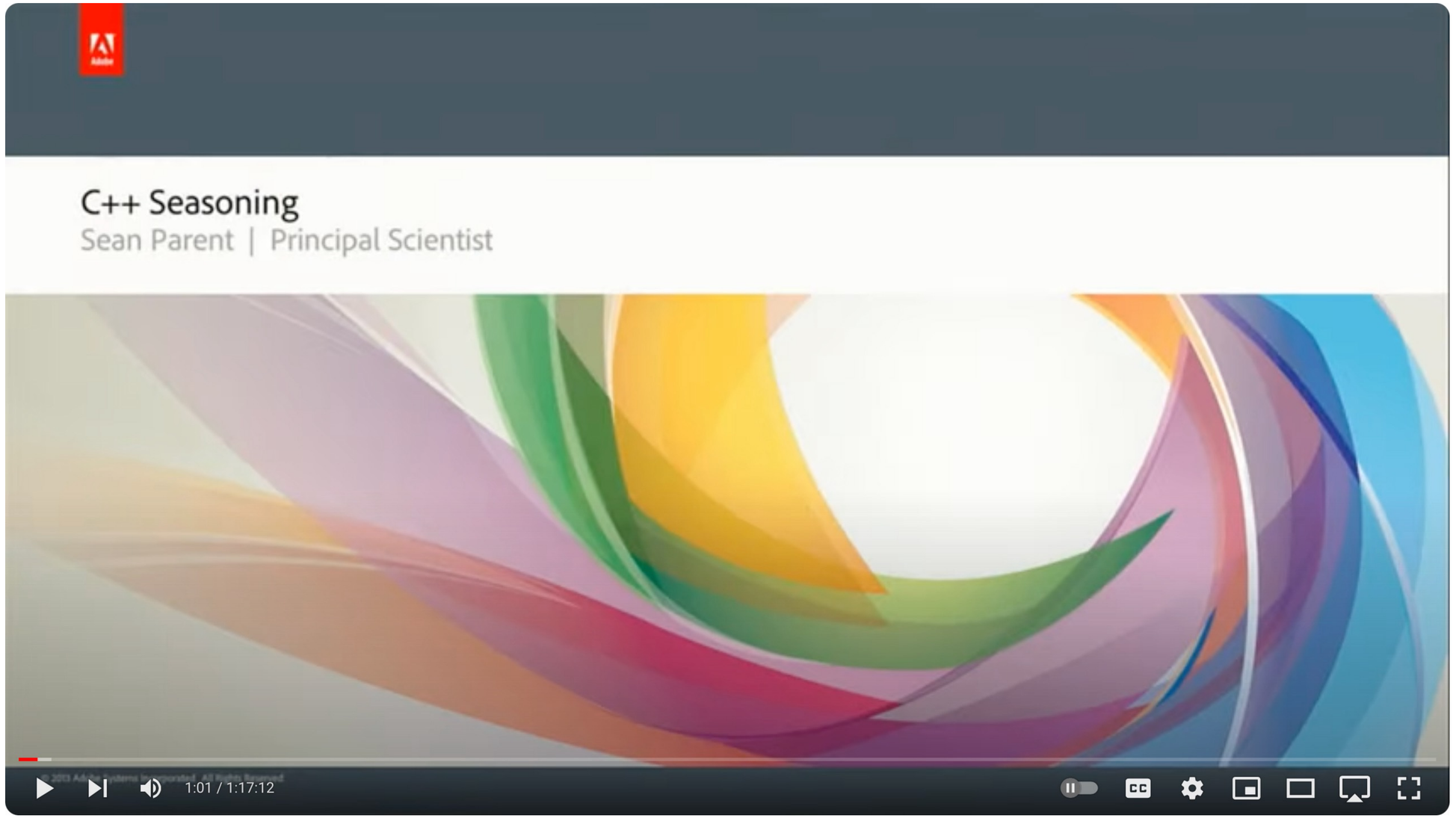
(Tony Van Eerd, @tvaneerd via Twitter)

How to Learn More About Algorithms

By now, I just love these algorithms! But there are so many, more than 200 in C++20. How can I learn more about them?

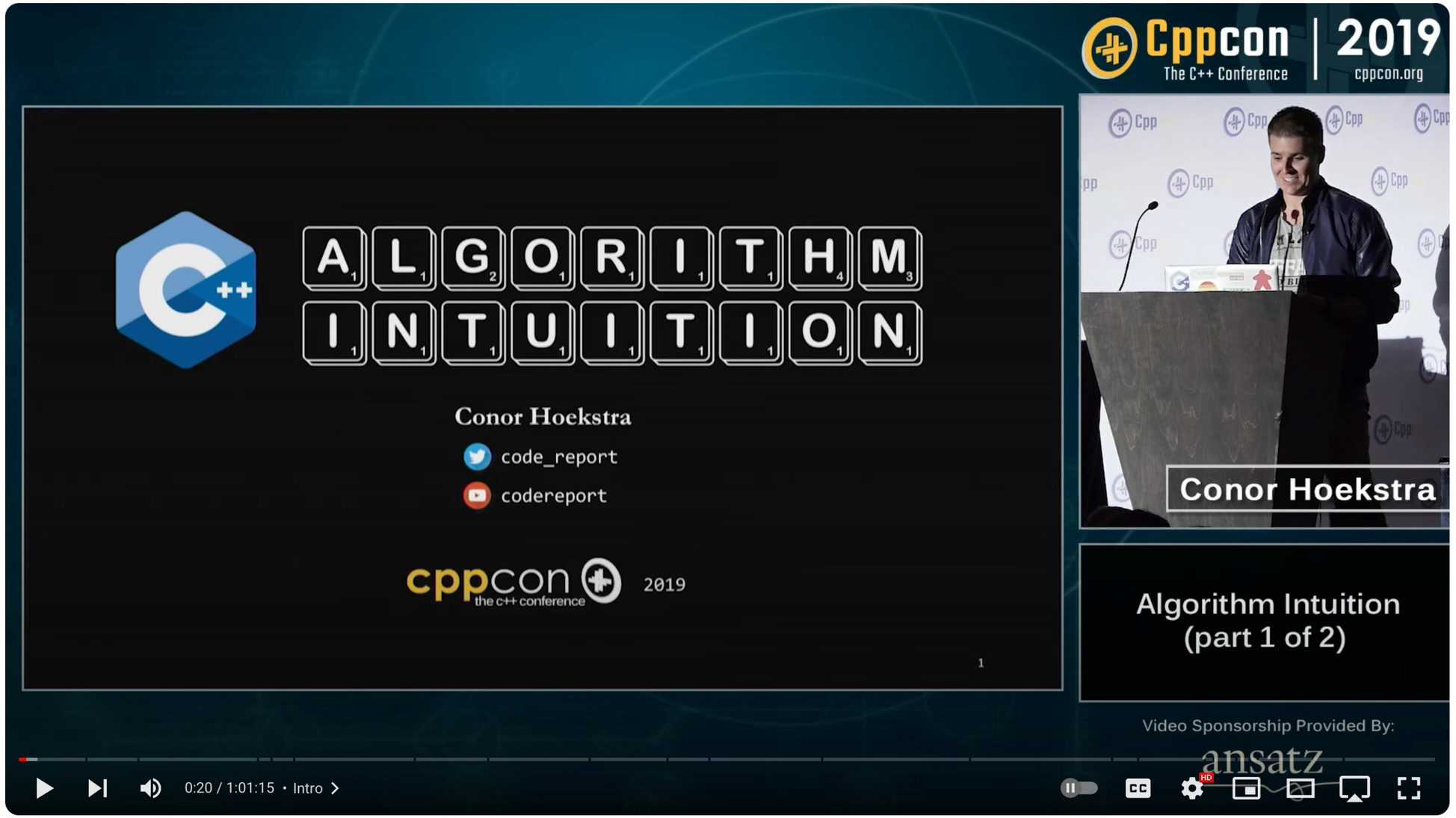
There are a couple of great talks to watch …

How to Learn More About Algorithms



How to Learn More About Algorithms

How to Learn More About Algorithms



Guidelines and Take-Aways

Don’t be like Bart and write all your for/while loops manually. **Prefer to use algorithms!**

Ouch, this hurts a little, but I see that you’re right! Algorithms are **simple**, **elegant**, **fast** and **remove so much potential for errors**!

Prefer to use the C++20 ranges algorithms.

Remember the conventions and the few pitfalls.

Learn more about algorithms.

One Last Question


```
int main()
{
    std::vector<Person> table =
        { Person{ "Homer",      "Simpson",    38 }
        , Person{ "Marge",      "Simpson",    34 }
        , Person{ "Bart",       "Simpson",    10 }
        , Person{ "Lisa",       "Simpson",     8 }
        , Person{ "Maggie",     "Simpson",     1 }
        , Person{ "Hans",       "Moleman",    33 }
        , Person{ "Ralph",      "Wiggum",      8 }
        , Person{ "Milhouse",   "Van Houten", 10 }
        , Person{ "Ned",        "Flanders",   60 }
        , Person{ "Jeff",       "Albertson",  45 }
        , Person{ "Montgomery", "Burns",     104 }
        , /* many more Simpson characters */ };

    // ...

    return EXIT_SUCCESS;
}
```

One Last Question


```
int main()
{
    std::vector<Person> table =
        { Person{ "Homer",      "Simpson",     38 }
        , Person{ "Marge",      "Simpson",     34 }
        , Person{ "Bart",       "Simpson",     10 }
        , Person{ "Lisa",       "Simpson",      8 }
        , Person{ "Maggie",     "Simpson",      1 }
        , Person{ "Hans",       "Moleman",     33 }
        , Person{ "Ralph",      "Wiggum",       8 }
        , Person{ "Milhouse",   "Van Houten", 10 }
        , Person{ "Ned",        "Flanders",    60 }
        , Person{ "Jeff",       "Albertson",   45 }
        , Person{ "Montgomery", "Burns",      104 }
        , /* many more Simpson characters */ };

    // ...

    return EXIT_SUCCESS;
}
```

Who is Jeff Albertson?

Jeff Albertson - The Comic Book Guy

Back To Basics
Algorithms

KLAUS IGLBERGER