Open4Tech Summer School 2020

C++17/20 STL<Essentials>
Code gold, not trash
RESTful APIs

TikTok hand challenge recognition using Javascript
Web Development Basics
Processing web data with XML and XSLT

24 iunie - 10 iulie 2020
http://inf.ucv.ro/~summer-school/

Departamentul de Informatică

CAPHYON

syncro soft

Society for Computing Technologies

Open4Tech Summer School 2020

	Luni	Marti	Miercuri	Joi	Vineri
	22 iunie	23 iunie	24 iunie	25 iunie	26 iunie
2-4pm			C++17/20 STL<Essentials>	C++17/20 STL<Essentials>	C++17/20 STL<Essentials>
4-6pm			Code gold, not trash	Web Development Basics	Web Development Basics
	29 iunie	30 iunie	1 iulie	2 iulie	3 iulie
2-4pm	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript		
4-6pm	RESTful APIs	RESTful APIs	RESTful APIs	RESTful APIs	
	6 iulie	7 iulie	8 iulie	9 iulie	10 iulie
2-4pm					
4-6pm			Processing web data with XML and XSLT	Processing web data with XML and XSLT	

http://inf.ucv.ro/~summer-school/

C++17/20

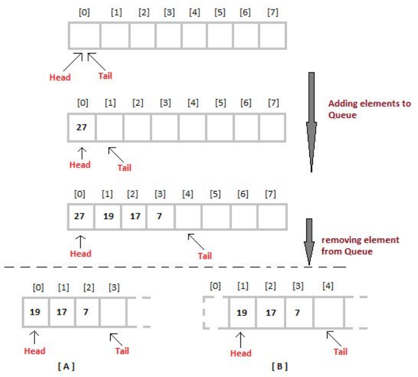
STL<Essentials>

Victor Ciura - Technical Lead

CAPHYON

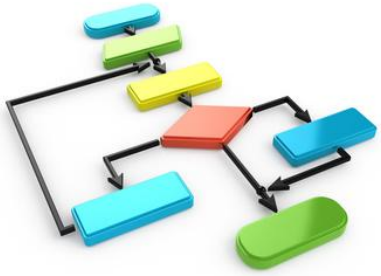
http://inf.ucv.ro/~summer-school/

Containers and Iterators



STL Function Objects and Utilities

STL Algorithms Principles and Practice



Fun with STL algorithms: What does it print ?

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <numeric>
#include <vector>

#define 🚗 const auto
#define 🤹 std::accumulate
#define 🔢 std::sort
#define 🏳️ empty()
#define 📏 size()
#define 👉 begin()
#define 👈 end()
#define 🔱 []

using 📜 = std::string;
template<typename T>
using 💰 = std::vector<T>;
```

```cpp
🚗 🛠 = "algorithms";
🚗 🔗 = " ";
🚗 💔 = "really love";
🚗 🎶 = "!";

🚗 🏹(💰<📜> & 📦)
{
    🔢(📦.👉, 📦.👈, 🔱(🚗 & 💚, 🚗 & 💜)
    {
        return 💚.📏 < 💜.📏;
    });

    return 🤹(📦.👉, 📦.👈, 📜(),
              🔱(🚗 & 🤓🤓, 🚗 & 🙄)
              {
                  return (🤓🤓.🏳️ ? 🙄 : (🤓🤓 + 🔗)) + 🙄;
              });
}

int main()
{
    💰<📜> 😃😃😃 = { 🛠, 💔, 🎶 };
    std::cout << 🏹(😃😃😃) << std::endl;
    return 0;
}
```

But first...

STL Background

STL and Its Design Principles

Generic Programming

- algorithms are associated with a **set of common properties**

  Eg. op { +, *, min, max } => associative operations => reorder operands

  => parallelize + reduction (std::accumulate)

- find the most general representation of algorithms (**abstraction**)
- exists a **generic algorithm** behind every WHILE or FOR loop
- natural extension of 4,000 years of **mathematics**

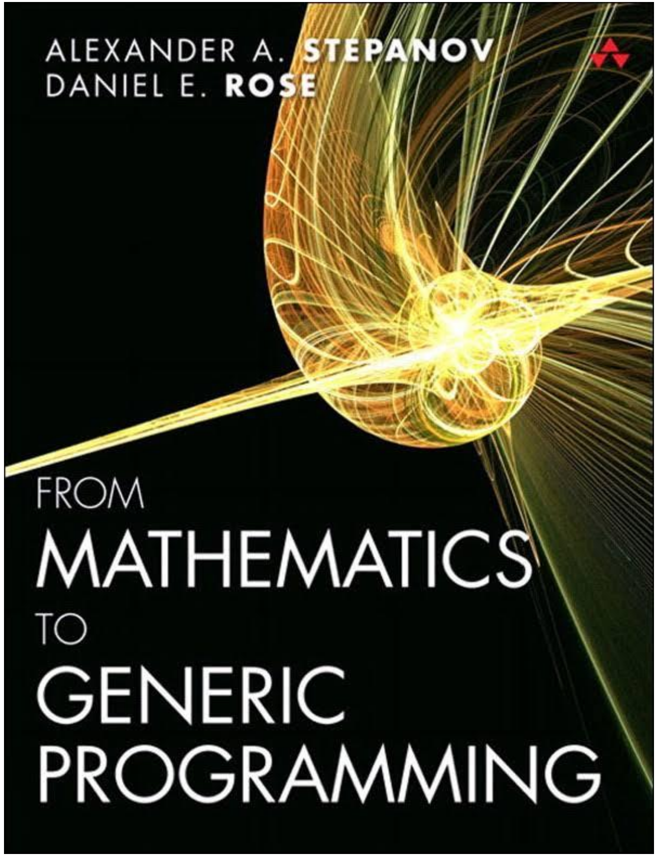
Alexander Stepanov (2002),

https://www.youtube.com/watch?v=COuHLky7E2Q

STL and Its Design Principles

Generic Programming

- Egyptian multiplication ~ 1900-1650 BC
- Ancient Greek number theory
- Prime numbers
- Euclid’s GCD algorithm
- Abstraction in mathematics
- Deriving generic algorithms
- Algebraic structures
- Programming concepts
- Permutation algorithms
- Cryptology (RSA) ~ 1977 AD



STL Data Structures

- they implement whole-part semantics (copy is deep - members)
- 2 objects never intersect (they are separate entities)
- 2 objects have separate lifetimes
- STL algorithms work only with ***Regular*** data structures
- **Semiregular** = *Assignable* + *Constructible* (both *Copy* and *Move* operations)
- **Regular** = Semiregular + *EqualityComparable*
- => STL assumes **equality** is always defined (at least, equivalence relation)

Video: "Regular Types and Why Do I Care"

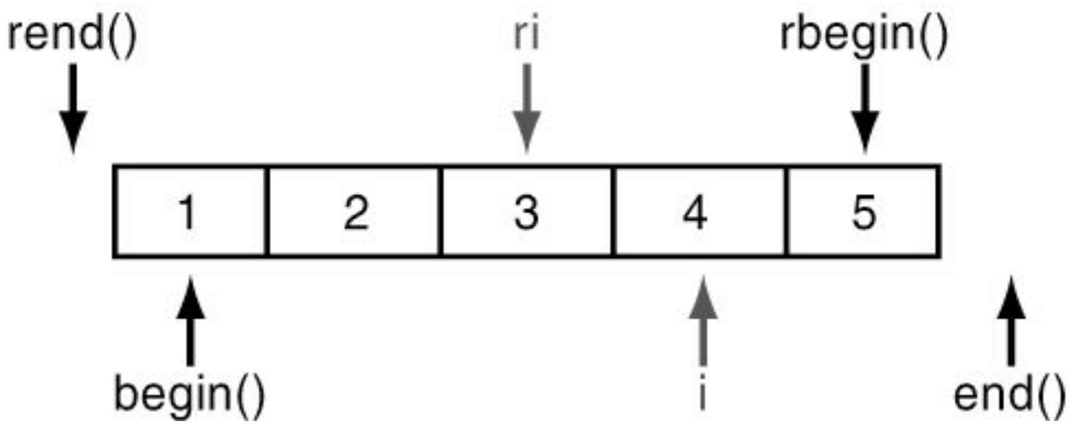
STL Iterators

- **Iterators** are the mechanism that makes it possible to *decouple* **algorithms** from **containers**.
- **Algorithms** are *template functions* parameterized by the **type of iterator**, so they are not restricted to a single type of container.
- An iterator represents an abstraction for a memory address (**pointer**).
- An iterator is an **object** that can iterate over elements in an STL container or range.
- All containers provide iterators so that algorithms can access their elements in a ***standard*** way.

STL Iterators

Ranges

- STL ranges are always semi-open intervals: `[b, e)`
- Get the beginning of a range/container: `v.begin();` or `begin(v);`
- You can get a reference to the first element in the range by: `*v.begin();`
- You cannot dereference the iterator returned by: `v.end();` or `end(v);`



STL Iterators

Iterate a collection (range-for)

```
std::array<int, 5> v = {2, 4, 6, 8, 10};

for(auto it = v.begin(); it != v.end(); ++it)  { … }


auto it  = v.begin();
auto end = v.end();
for(; it != end; ++it)  { … }


for(auto val : v) { … }
```

https://cppinsights.io

C-style iteration vs STL Iterators

📋 Reuse existing code so that is prints letters in reverse order.

The C way

CODE

```
vector<char> letters = { 'S', 'T', 'L' };
for (unsigned int n = 0; n < letters.size(); ++n)
  cout << letters[n] << " ";
```

```
vector<char> letters = { 'L', 'T', 'S' };
for (unsigned int i = letters.size(); i >= 0; --i)
  cout << letters[n] << " ";
```

Can you spot any issues with this code?

OUTPUT

S T L

???

Out of bounds memory error
Because of signed integer underflow

Out of bounds memory error.
We need the decrement operator

Introducing a bug. We're skipping the 'S'

Off-by-one error. We need to start from size() - 1

Old code forgotten during refactoring.
Compiler will catch this

C-style iteration vs *STL Iterators*

📋Reuse existing code so that is prints letters in reverse order.

The C way

CODE | OUTPUT

```
vector<char> letters = { 'S', 'T', 'L' };
for (unsigned int n = 0; n < letters.size(); ++n)
  cout << letters[n] << " ";
```

S T L

```
vector<char> letters = { 'L', 'T', 'S' };
for (unsigned int i = letters.size() - 1; i >= 0; --i)
{
  cout << letters[i] << " ";
  if (i == 0) break;
}
```

S T L

C-style iteration* vs *STL Iterators

📋 Reuse existing code so that is prints letters in reverse order.

The `STL Iterators` way

CODE

```
vector<char> letters = { 'S', 'T', 'L' };
for (auto i = letters.begin(), ei = letters.end(); i != ei; ++i)
  cout << *i << " ";
```

```
vector<char> letters = { 'L', 'T', 'S' };
for (auto it = nrs.rbegin(), endIt = nrs.rend(); it!= endIt; ++it)
  cout << *it << " ";
```

Can you spot any issues with this code?

Old code forgotten during refactoring.
Induction variable has different name

OUTPUT

S T L

S T L

C-style iteration vs *STL Iterators*

📋 Reuse existing code so that is prints letters in reverse order.

The `range-for` way

CODE

```
vector<char> letters = { 'S', 'T', 'L' };
for (auto letter : letters)
  cout << letter << " ";
```

OUTPUT

S T L

```
vector<char> letters = { 'L', 'T', 'S' };
for (auto letter : reverse(letters))
  cout << letter << " ";
```

S T L

✓ No issues here

ℹ `reverse()` is an iterator adapter, which we'll introduce shortly

Iterate a collection in reverse order

```
std::vector<int> values;
```

C style:

```
for (int i = values.size() - 1; i >= 0; --i)
  cout << values[i] << endl;
```

C++98:

```
for(vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) { … }
```

STL + Lambdas:

```
for_each( values.rbegin()), values.rend(),
          [](const string & val) { cout << val << endl; } );
```

Modern C++ range-for, using *iterator adapter*:

```
for ( auto & val : reverse(values) ) { cout << val << endl; }
```

Iterate a collection in reverse order C++ 20

C++ 20 ranges coming *soon* to your compiler of choice:

```
for (auto & val :  ranges::reverse_view(values))
{
  cout << val << endl;
}
```

C++ 20

C++ 20 ranges are a *major* feature to the language

Here's *a peek of what they enable:*

```
vector<int> ints { 0, 1, 2, 3, 4, 5};
auto isEven   = [](int i) { return i % 2 == 0; };
auto toSquare = [](int i) { return i * i; };

for (int i : ints | views::filter(isEven) | views::transform(toSquare))
{
  std::cout << i << ' ';
}
```

PRINTS: **0 4 8**

Iterator Adaptors

Iterate a collection in reverse order

```
namespace detail
{
  template <typename T>
  struct reversion_wrapper
  {
    T & mContainer;
  };
}

/**
 * Helper function that constructs
 * the appropriate iterator type based on ADL.
 */
template <typename T>
detail::reversion_wrapper<T> reverse(T && aContainer)
{
  return { aContainer };
}
```

Iterator Adaptors

Iterate a collection in reverse order

```
namespace std
{
  template <typename T>
  auto begin(detail::reversion_wrapper<T> aRwrapper)
  {
    return rbegin(aRwrapper.mContainer);
  }

  template <typename T>
  auto end(detail::reversion_wrapper<T> aRwrapper)
  {
    return rend(aRwrapper.mContainer);
  }
}
```

Iterator Adaptors

Homework:

Iterate through an associative container `keys` or `values`

first second first second

```
std::unordered_map<string, int> weights; // container value types are <key, value> pairs

// fill some weights in the map and compute the total
int totalWeight = 0;
for ( auto & val : iterate_second(weights) ) { totalWeight += val; }
```

Using the same technique shown for `reverse()` iteration adaptor, implement this helpful `iterate_second()` adaptor.

Can you replace the *range-for* with an STL algorithm ?
https://en.cppreference.com/w/cpp/algorithm

Email solutions to: open4tech@caphyon.com

Function Objects Basics

```
template<class InputIt, class UnaryFunction>
void std::for_each( InputIt first, InputIt last, UnaryFunction func )
{
  for(; first != last; ++first)
    func( *first );
}

struct Printer // our custom functor for console output
{
  void operator()(const std::string & str)
  {
    std::cout << str << std::endl;
  }
};

std::vector<std::string> vec = { "STL", "function", "objects", "rule" };

std::for_each(vec.begin(), vec.end(), Printer());
```

λ Lambda Functions

```
struct Printer // our custom functor for console output
{
  void operator()(const string & str)
  {
    cout << str << endl;
  }
};

std::vector<string> vec = { "STL", "function", "objects", "rule" };

std::for_each(vec.begin(), vec.end(), Printer());



// using a lambda

std::for_each(vec.begin(), vec.end(),
              [](const string & str) { cout << str << endl; });
```

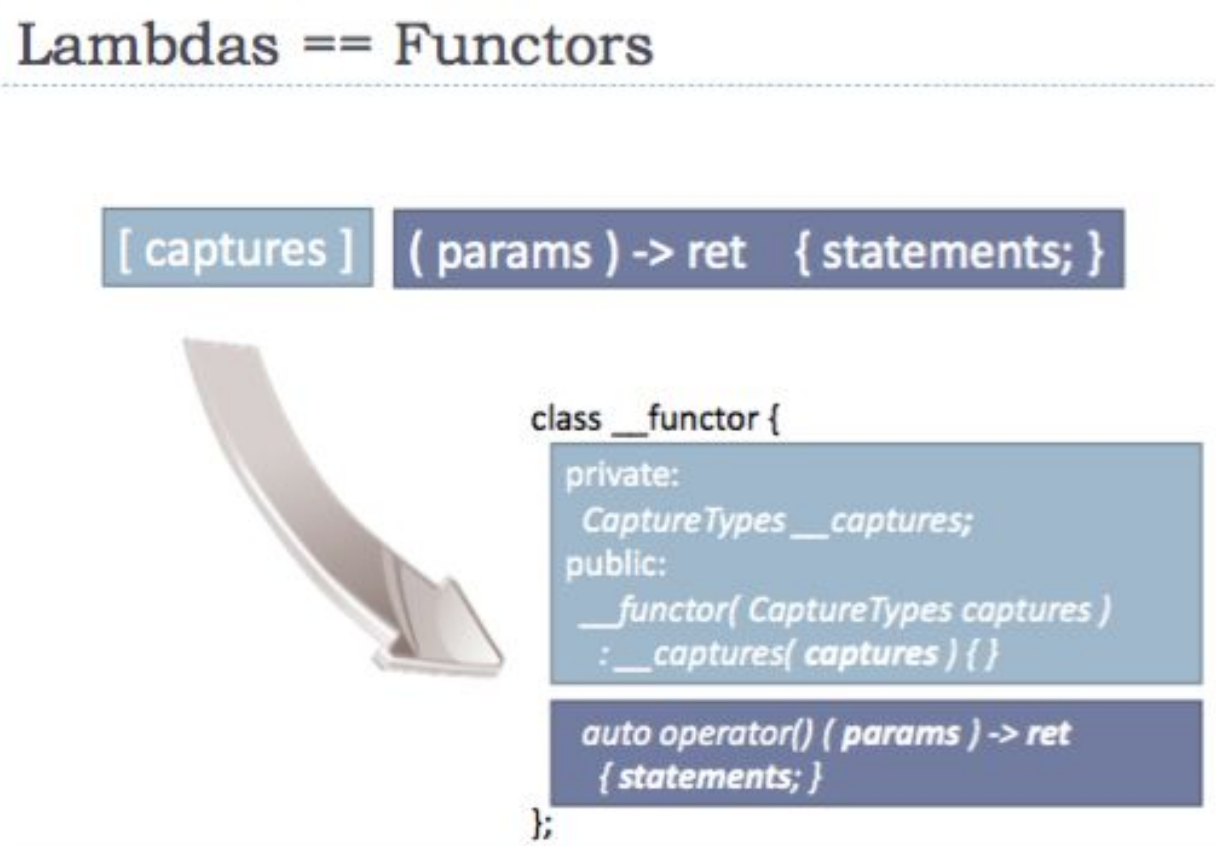
λ Lambda Functions

```
[ capture-list ] ( params ) mutable(optional) -> ret { body }

[ capture-list ] ( params ) -> ret { body }

[ capture-list ] ( params ) { body }

[ capture-list ] { body }
```

Capture list can be passed as follows :

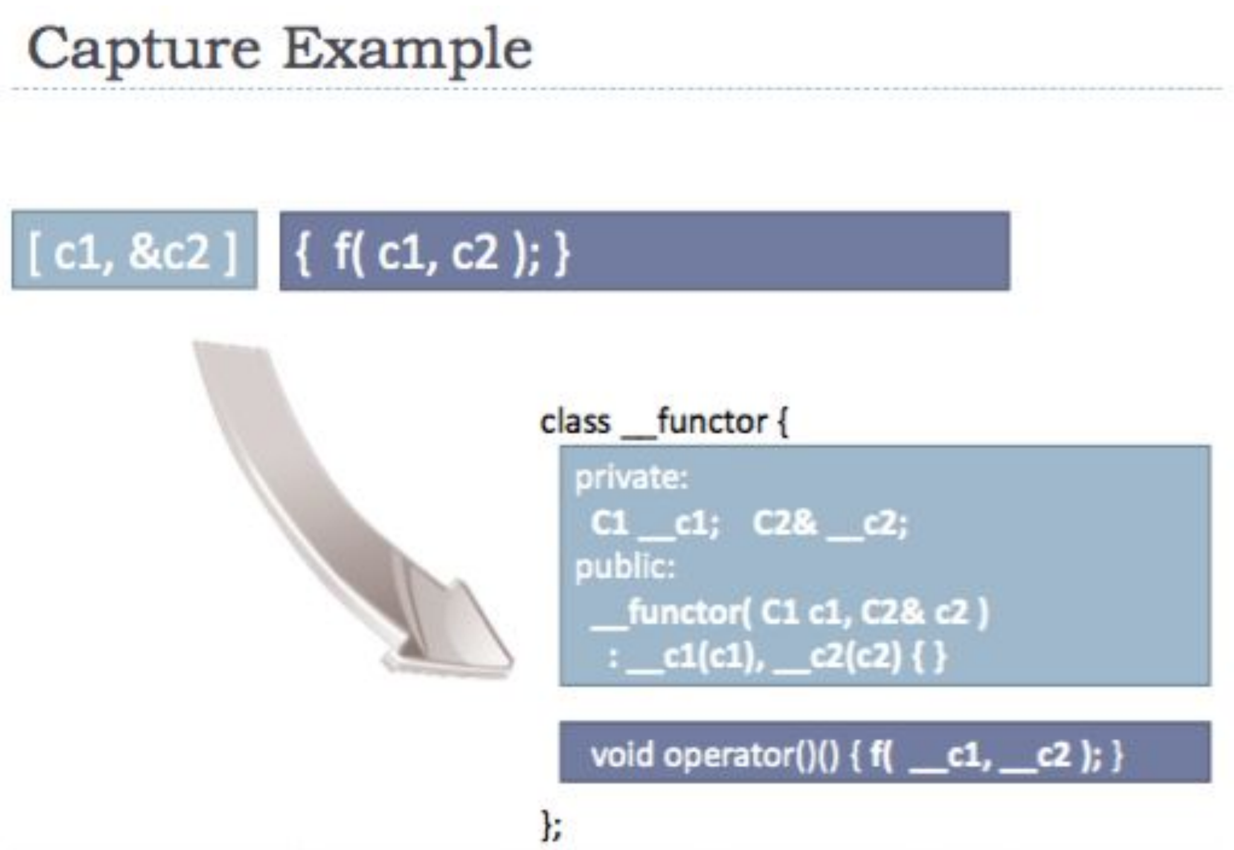
- **[a, &b]** where **a** is captured by ***value*** and **b** is captured by ***reference***.
- **[this]** captures the **this** pointer by ***value***
- **[&]** captures all automatic variables **used** in the body of the lambda by ***reference***
- **[=]** captures all automatic variables **used** in the body of the lambda by ***value***
- **[]** captures ***nothing***

Anatomy of A Lambda



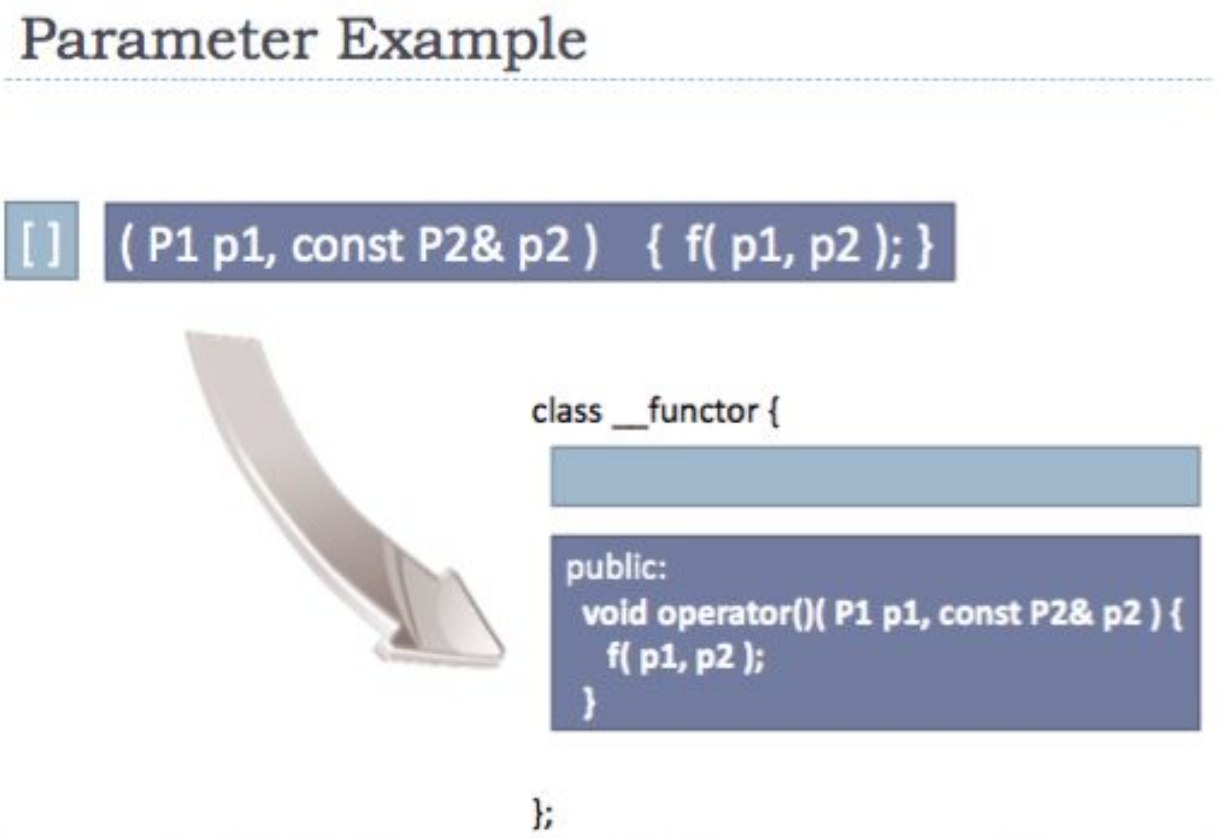
credit: Herb Sutter - “*Lambdas, Lambdas Everywhere*”
https://www.youtube.com/watch?v=rcgRY7sOA58

Anatomy of A Lambda



credit: Herb Sutter - “*Lambdas, Lambdas Everywhere*”
https://www.youtube.com/watch?v=rcgRY7sOA58

Anatomy of A Lambda



credit: Herb Sutter - "*Lambdas, Lambdas Everywhere*"
https://www.youtube.com/watch?v=rcgRY7sOA58

Lambda Functions

```
std::list<Person> members = {...};
unsigned int minAge = GetMinimumAge();
members.remove_if( [minAge](const Person & p) { return p.age < minAge; } );
```

https://cppinsights.io

Lambda Functions

```
std::list<Person> members = {...};
unsigned int minAge = GetMinimumAge();
members.remove_if( [minAge](const Person & p) { return p.age < minAge; } );
```

⇩

```
// compiler generated code:
struct Lambda_247
{
  Lambda_247(unsigned int _minAge) : minAge(_minAge) {}
  bool operator()(const Person & p) { return p.age < minAge; }
  unsigned int minAge;
};

members.remove_if( Lambda_247(minAge) );
```

https://cppinsights.io

Prefer Function Objects or Lambdas to Free Functions

```
vector<int> v = { … };

bool GreaterInt(int i1, int i2) { return i1 > i2; }

sort(v.begin(), v.end(), GreaterInt); // pass function pointer

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

sort(v.begin(), v.end(), [](int i1, int i2) { return i1 > i2; });
```

WHY ?

Function Objects and Lambdas leverage `operator()` **inlining**
vs.
indirect **function call** through a *function pointer*

This is the main reason ***std::sort()*** *outperforms* ***qsort()*** *from* ***C****-runtime by at least 500% in typical scenarios, on large collections.*

STL Algorithms - Principles and Practice

“Prefer algorithm calls to hand-written loops.”

Scott Meyers, "Effective STL"

Why prefer to use (STL) algorithms?

Goal: No Raw Loops {}

Sean Parent *- C++ Seasoning, 2013*

Whenever you want to write a **`for/while`** loop:

```
for(int i = 0; i < v.size(); ++i) { … }
```

Put the Mouse Down and Step Away from the Keyboard !

Burk Hufnagel

Why prefer to use (STL) algorithms?

Correctness

Fewer opportunities to write bugs like:

- iterator invalidation
- copy/paste bugs
- iterator range bugs
- loop continuations or early loop breaks
- guaranteeing loop invariants
- issues with algorithm logic

Code is a liability: maintenance, people, knowledge, dependencies, sharing, etc.

More code => more bugs, more test units, more maintenance, more documentation

Why prefer to use (STL) algorithms?

Code Clarity

- Algorithm **names** say what they do.
- Raw “for” loops don’t (without reading/understanding the whole body).
- We get to program at a higher level of **abstraction** by using well-known **verbs** (find, sort, remove, count, transform).
- A piece of code is **read** many more times than it’s **modified**.
- **Maintenance** of a piece of code is greatly helped if all future programmers understand (with confidence) what that code does.

Is simplicity a good goal ?

- Simpler code is more **readable** code
- Unsurprising code is more **maintainable** code
- Code that moves complexity to **abstractions** often has **less bugs**
  - corner cases get covered by the **library** writer
  - **RAII** ensures nothing is forgotten
- Compilers and libraries are often much better than you (**optimizing**)
  - they’re guaranteed to be better than someone who’s not measuring

Kate Gregory, *“It’s Complicated”*, Meeting C++ 2017

What does it mean for code to be simple ?

- Easy to **read**
- Understandable and **expressive**
- Usually, **shorter** means simpler (but not always)
- **Idioms** can be simpler than they first appear (because they are recognized)

Kate Gregory, *“It’s Complicated”*, Meeting C++ 2017

Simplicity is Not Just for Beginners

- Requires **knowledge**
  - language / syntax
  - idioms
  - what can go wrong
  - what might change some day
- Simplicity is an act of **generosity**
  - to others
  - to future you
- <u>Not</u> about **leaving out**
  - *meaningful names*
  - error handling
  - testing
  - documentation

Kate Gregory, *“It’s Complicated”*, Meeting C++ 2017

Why prefer to use (STL) algorithms?

Modern C++ (ISO 14/17/20 standards)

- Modern C++ adds more useful algorithms to the STL library.
- Makes existing algorithms much easier to use due to simplified language syntax and lambda functions (closures).

```
for(vector<string>::iterator it = v.begin(); it != v.end(); ++it)  { … }

for(auto it = v.begin(); it != v.end(); ++it)  { … }

for(auto it = v.begin(), end = v.end(); it != end; ++it)  { … }

std::for_each(v.begin(), v.end(), [](const auto & val) { … });

for(const auto & val : v) { … }
```

Why prefer to use (STL) algorithms?

Performance / Efficiency

What's the difference?

- Vendor implementations are highly **tuned** (most of the time).
- Avoid some unnecessary temporary copies (leverage **move** operations for objects).
- Function helpers and functors are **inlined** away (no abstraction penalty).
- Compiler optimizers can do a better job without worrying about **pointer aliasing**

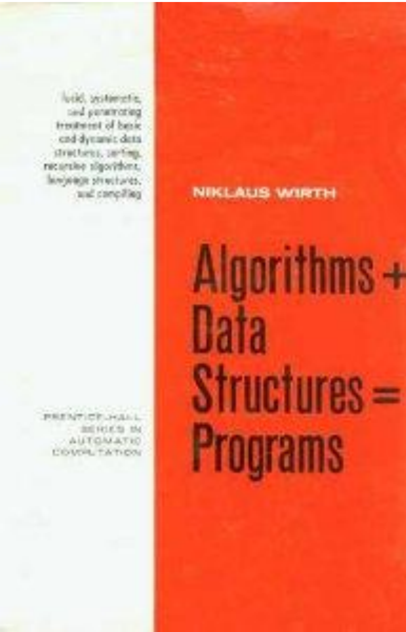
  (auto-vectorization, auto-parallelization, loop unrolling, dependency checking, etc.).

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



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 dependant** on one another.

Optimization

Strategy:

1. **Identification**: profile the application and identify the worst performing parts.
2. **Comprehension**: understand what the code is trying to achieve and why it is slow.
3. **Iteration**: change the code based on step 2 and then re-profile; repeat until fast enough.

Very often, code becomes a bottleneck for one of four reasons:

- It's being called too often.
- It's a bad choice of algorithm: O(n^2) vs O(n), for example.
- It's doing unnecessary work or it is doing necessary work too frequently.
- The data is bad: either too much data or the layout and access patterns are bad.

Don't trust your instinct.

Always Benchmark !

Open4Tech Summer School 2020

C++17/20 STL<Essentials>
Code gold, not trash
RESTful APIs

TikTok hand challenge recognition using Javascript
Web Development Basics
Processing web data with XML and XSLT

24 iunie - 10 iulie 2020
http://inf.ucv.ro/~summer-school/

DEPARTAMENTUL DE INFORMATICĂ

CAPHYON

syncro soft

Society for Computing Technologies

Open4Tech Summer School 2020

	Luni	Marti	Miercuri	Joi	Vineri
	22 iunie	23 iunie	24 iunie	25 iunie	26 iunie
2-4pm			C++17/20 STL<Essentials>	C++17/20 STL<Essentials>	C++17/20 STL<Essentials>
4-6pm			Code gold, not trash	Web Development Basics	Web Development Basics
	29 iunie	30 iunie	1 iulie	2 iulie	3 iulie
2-4pm	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript		
4-6pm	RESTful APIs	RESTful APIs	RESTful APIs	RESTful APIs	
	6 iulie	7 iulie	8 iulie	9 iulie	10 iulie
2-4pm					
4-6pm			Processing web data with XML and XSLT	Processing web data with XML and XSLT	

http://inf.ucv.ro/~summer-school/

C++17/20

STL<Essentials>

Victor Ciura - Technical Lead

http://inf.ucv.ro/~summer-school/

Introduction to Algorithms

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

March, 1990

A classic, priceless book

30 years after publication

https://www.amazon.co.uk/Introduction-Algorithms

Homework

Fun with squares^2

A deceptively simple problem

"You have a vector of integers sorted in non-decreasing order;
compute the squares of each number, also in sorted non-decreasing order."

```
-5 -2 0 6 8   =>  0 4 25 36 64
```

A *naive* solution:

```
for (auto & e : vec)
  e *= e;
std::sort(begin(vec), end(vec));
```

- Can you implement a solution avoiding the **cost of sorting** ?
- Explore **in-place** solutions (like the naive one), as well as using a **separate** vector for the result
- Can you trade **space** for **speed** in this example ?
- What STL **algorithms** can you identify as useful here ?
- Can you find a solution that is **linear** or *better* ?
- Tip: take advantage of all the **constraints** of the problem and the initial conditions

Email solutions to open4tech@caphyon.com

Recap<T>

STL and Its Design Principles

Generic Programming

- algorithms are associated with a **set of common properties**

  Eg. op { +, *, min, max } => associative operations => reorder operands

  => parallelize + reduction (std::accumulate)

- find the most general representation of algorithms (**abstraction**)
- exists a **generic algorithm** behind every WHILE or FOR loop
- natural extension of 4,000 years of **mathematics**

Alexander Stepanov (2002),

https://www.youtube.com/watch?v=COuHLky7E2Q

Generic Programming Drawbacks

- abstraction penalty
- implementation in the interface
- early binding
- horrible error messages (*no formal specification* of interfaces, **yet**)
- duck typing
- algorithm could work on some data types, but fail to work/compile on some other new data structures (different iterator category, no copy semantics, etc)

We need to fully specify **requirements** on algorithm types => `Concepts`

Named Requirements

Some examples from **STL**:

- `DefaultConstructible, MoveConstructible, CopyConstructible`
- `MoveAssignable, CopyAssignable,`
- `Destructible`
- `EqualityComparable, LessThanComparable`
- `Predicate, BinaryPredicate`
- `Compare`
- `FunctionObject`
- `Container, SequenceContainer, ContiguousContainer, AssociativeContainer`
- `Iterator`
  - `InputIterator, OutputIterator`
  - `ForwardIterator, BidirectionalIterator, RandomAccessIterator`

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

Named Requirements

Named requirements are used in the normative text of the C++ standard to define the expectations of the standard library.

Some of these requirements were formalized in **C++20** using `concepts`.

If you're not using C++20 yet, the burden is on YOU to ensure that library templates are instantiated with template arguments that satisfy these requirements.

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

C++20

What Is A Concept, Anyway ?

Formal specification of concepts makes it possible to **verify** that template arguments satisfy the expectations of a template or function during overload resolution and template specialization (requirements).

Each concept is a **predicate**, evaluated at *compile time*, and becomes a part of the interface of a template where it is used as a constraint.

https://en.cppreference.com/w/cpp/language/constraints

C++20

What Is A Concept, Anyway ?

The whole STL has been **conceptified**, starting with C++20

```
template< class InputIt, class UnaryPredicate >
constexpr bool any_of( InputIt first, InputIt last, UnaryPredicate pred );
```

std::ranges::

```
template< std::input_iterator I, std::sentinel_for<I> S,
          class Proj = std::identity,
          std::indirect_unary_predicate<std::projected<I, Proj>> Pred >
constexpr bool any_of( I first, S last, Pred pred, Proj proj = {} );
```

constraints

https://en.cppreference.com/w/cpp/language/constraints

What's the Practical Upside ?

If I'm not a library writer 🤓,
Why Do I Care ?

What's the Practical Upside ?

Using STL algorithms & data structures

Designing & exposing your own vocabulary types
(interfaces, APIs)

Compare Concept

Why is this one special ?
Because ~50 STL facilities (algorithms & data structures) expect a *Compare* type.

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

Concept relations:

Compare << *BinaryPredicate* << *Predicate* << *FunctionObject* << *Callable*

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`

https://en.wikipedia.org/wiki/Partially_ordered_set **{ partial ordering }**

Compare Examples

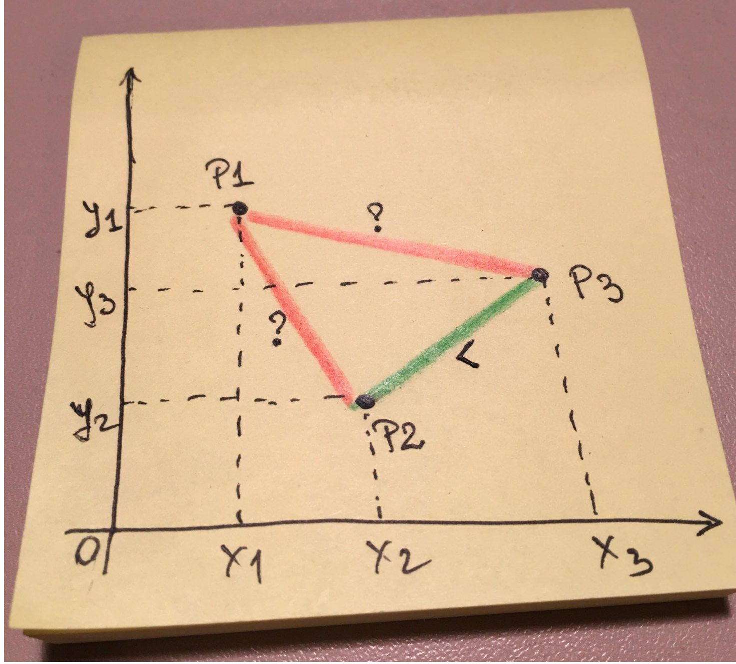
```
vector<string> v = { ... };

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

```
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



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
```

`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

Partial ordering relationship: *Irreflexivity* + *Antisymmetry* + *Transitivity*

Strict weak ordering relationship: ***Partial ordering*** + *Transitivity of Equivalence*

Total ordering relationship: ***Strict weak ordering*** + **equivalence** must be the same as **equality**

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`

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

Compare Examples

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

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

Is this a good Compare predicate for 2D points ?

Compare Examples

```
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;
});
```

Is this a 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.
(in our example 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”*.

The STL implements dictionary ordering in at least three places:

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

std::tuple< ... Types> - generalization of pair

std::lexicographical_compare() algorithm

- Two ranges are compared element by element
- The first mismatching element defines which range is lexicographically *less* or *greater* than the other
- ...

C++20

Compare

The **Spaceship** has landed !

`operator <=>`

- Consistent comparison
- Relationship strength (comparison category traits):
    - `strong_ordering`
    - `weak_ordering`
    - `partial_ordering`
    - `strong_equality`
    - `weak_equality`
- STL implementation for containers & utility classes
- Examples

Jonathan Müller “Using C++20's Three-way Comparison <=>”
https://www.youtube.com/watch?v=8jNXy3K2Wpk

STL Algorithms - Principles and Practice

“Show me the code”

A common task...

Remove elements matching a predicate.

Given:

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };
```

How do we remove all **even** numbers ?

A common task...

Remove elements matching a predicate.

https://en.cppreference.com/w/cpp/container/vector/erase

```
iterator vector::erase(const_iterator first, const_iterator last);
```

https://en.cppreference.com/w/cpp/algorithm/remove

```
template< class ForwardIt, class UnaryPredicate >
ForwardIt std::remove_if(ForwardIt first, ForwardIt last, UnaryPredicate p);
```

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };

v.erase( std::remove_if(v.begin(), v.end(),
                        [] (int i) { return (i & 1) == 0; }),
         v.end() );
```

How do you think this works ?

"remove_if() moves all the elements you want to remove to the **end** of the vector, then the erase gets rid of them."

```
v = { 1, 3, 5, 7, 2, 4, 6 }
```

WRONG !

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };

v.erase( std::remove_if(v.begin(), v.end(),
                        [] (int i) { return (i & 1) == 0; }),
         v.end() );
```

This **isn't** what `std::remove_if()` does !

If it did that – which is ***more work*** than it does – it would in fact be `std::partition()`.

What `std::remove()` does is move the elements that **won't** be removed **to the beginning**.

Erase-Remove Idiom

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6, 7 };

v.erase( std::remove_if(v.begin(), v.end(),
                        [] (int i) { return (i & 1) == 0; }),
         v.end() );
```

What about the elements at the **end** of the vector ?

GARBAGE !

They get *overwritten* in the process of `std::remove()` algorithm.

Before `erase()` is called: `v = { 1, 3, 5, 7, 5, 6, 7 }`

↑

`where the iterator **returned** by remove_if() points`

Prefer Member Functions To Similarly Named Algorithms

The following member functions are available for ***associative containers***:

```
-   .count()
-   .find()
-   .equal_range()
-   .lower_bound() // only for ordered containers
-   .upper_bound() // only for ordered containers
```

The following member functions are available for **`std::list`**

```
-   .remove()    .remove_if()
-   .unique()
-   .sort()
-   .merge()
-   .reverse()
```

These member functions are always **faster** than their similarly named generic algorithms.

Why? They can leverage the ***implementation details*** of the underlying data structure.

Prefer Member Functions To Similarly Named Algorithms

std::list<> specific algorithms

`std::sort()` doesn't work on lists (Why ?)
=> call `.sort()` member function

`.remove()` and `.remove_if()` don't need to use the **erase/remove idiom**.
They directly remove matching elements from the list.

`.remove()` and `.remove_if()` are more efficient than the generic algorithms,
because they just relink nodes with the need to copy or move elements.

Prefer Member Functions To Similarly Named Algorithms

```
std::set<string> s = {...}; // 1 million elements

// worst case: 1 million comparisons
// average:    ½ million comparisons
auto it = std::find(s.begin(), s.end(), "stl");
if (it != s.end()) {...}


// worst case: 40 comparisons
// average:    20 comparisons
auto it = s.find("stl");
if (it != s.end()) {...}
```

Why ?

Don’t Trust Your Intuition: Always Benchmark !

```
static void StdFind(benchmark::State & state)
{
  std::set<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.insert("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;

  }
}

BENCHMARK(StdFind);
```

```
static void SetFind(benchmark::State & state)
{
  std::set<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.insert("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = items.find("STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;
  }
}

BENCHMARK(SetFind);
```

http://quick-bench.com

Try increasing values for `COUNT_ELEM` : `500` >>> `500’000` >>> ...

Don't Trust Your Intuition: Always Benchmark !

```
static void ListFind(benchmark::State & state)
{
  std::list<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.push_back("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;

  }
}

BENCHMARK(ListFind);
```

```
static void VectorFind(benchmark::State & state)
{
  std::vector<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.push_back("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;
  }
}

BENCHMARK(VectorFind);
```

http://quick-bench.com

Try increasing values for `COUNT_ELEM` : 500 >>> 500'000 >>> ...

Binary search operations (on *sorted* ranges)

```
binary_search() // helper (incomplete interface - Why ?)
lower_bound()   // returns an iter to the first element not less than the given value
upper_bound()   // returns an iter to the first element greater than the certain value

equal_range() = { lower_bound(), upper_bound() }

// properly checking return value
auto it = lower_bound(v.begin(), v.end(), 5);
if ( it != v.end() && (*it == 5) )
{
  // found item, do something with it
}
else // not found, insert item at the correct position
{
  v.insert(it, 5);
}
```

*Why do we need to check the **value** we searched for ?*

Binary search operations (on *sorted* ranges)

Counting elements equal to a given value

```
vector<string> v = { … };  // sorted collection
size_t num_items = std::count(v.begin(), v.end(), "stl");
```

Instead of using `std::count()` generic algorithm, use **binary search** instead.

```
auto range = std::equal_range(v.begin(), v.end(), "stl");
size_t num_items = std::distance(range.first, range.second);
```

Open4Tech Summer School 2020

C++17/20 STL<Essentials>
Code gold, not trash
RESTful APIs

TikTok hand challenge recognition using Javascript
Web Development Basics
Processing web data with XML and XSLT

24 iunie - 10 iulie 2020
http://inf.ucv.ro/~summer-school/

Departamentul de Informatică

CAPHYON

syncro soft

Society for Computing Technologies

Open4Tech Summer School 2020

	Luni	Marti	Miercuri	Joi	Vineri
	22 iunie	23 iunie	24 iunie	25 iunie	26 iunie
2-4pm			C++17/20 STL<Essentials>	C++17/20 STL<Essentials>	C++17/20 STL<Essentials>
4-6pm			Code gold, not trash	Web Development Basics	Web Development Basics
	29 iunie	30 iunie	1 iulie	2 iulie	3 iulie
2-4pm	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript	TikTok hand challenge recognition using Javascript		
4-6pm	RESTful APIs	RESTful APIs	RESTful APIs	RESTful APIs	
	6 iulie	7 iulie	8 iulie	9 iulie	10 iulie
2-4pm					
4-6pm			Processing web data with XML and XSLT	Processing web data with XML and XSLT	

http://inf.ucv.ro/~summer-school/

C++17/20

STL<Essentials>

Victor Ciura - Technical Lead

CAPHYON

http://inf.ucv.ro/~summer-school/

STL,
To infinity and beyond …

STL was designed to be extended ...

Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
bool name_this_algorithm(Container & c, const Value & v)
{
  return std::find(begin(c), end(c), v) != end(c);
}
```

Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
void name_this_algorithm(Container & c, const Value & v)
{
  if ( std::find(begin(c), end(c), v) == end(c) )
    c.emplace_back(v);
}
```

Extend STL With Your Generic Algorithms

Eg.

```
template<class Container, class Value>
bool name_this_algorithm(Container & c, const Value & v)
{
  auto found = std::find(begin(c), end(c), v);
  if (found != end(v))
  {
    c.erase(found); // call 'erase' from STL container
    return true;
  }
  return false;
}
```

Consider Adding Range-based Versions of STL Algorithms

```
namespace range {   // our <algorithm_range.h> has ~150 wrappers for std algorithms

  template< class InputRange, class T > inline
  typename auto find(InputRange && range, const T & value)
  {
    return std::find(begin(range), end(range), value);
  }

  template< class InputRange, class UnaryPredicate > inline
  typename auto find_if(InputRange && range, UnaryPredicate pred)
  {
    return std::find_if(begin(range), end(range), pred);
  }

  template< class RandomAccessRange, class BinaryPredicate > inline
  void sort(RandomAccessRange && range, BinaryPredicate comp)
  {
    std::sort(begin(range), end(range), comp);
  }

}
```

Until C++20

Consider Adding Range-based Versions of STL Algorithms

Eg.

```
vector<string> v = { … };

auto it = range::find(v, "stl");
string str = *it;

auto chIt = range::find(str, 't');

auto it2 = range::find_if(v, [](const auto & val) { return val.size() > 5; });

range::sort(v);

range::sort(v, [](const auto & val1, const auto & val2)
                 { return val1.size() < val2.size(); } );
```

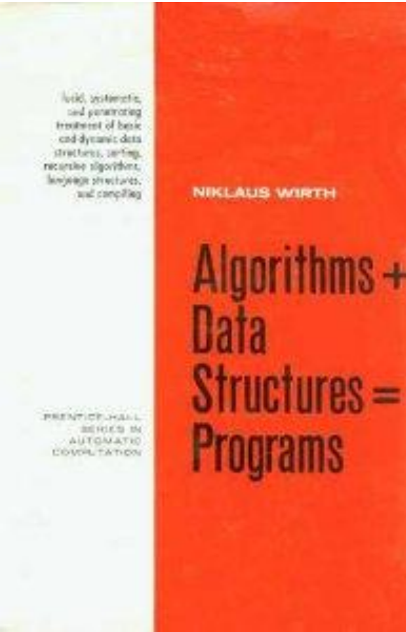
Until C++20

Efficiency   Press both buttons   Performance

Why do we care ?

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

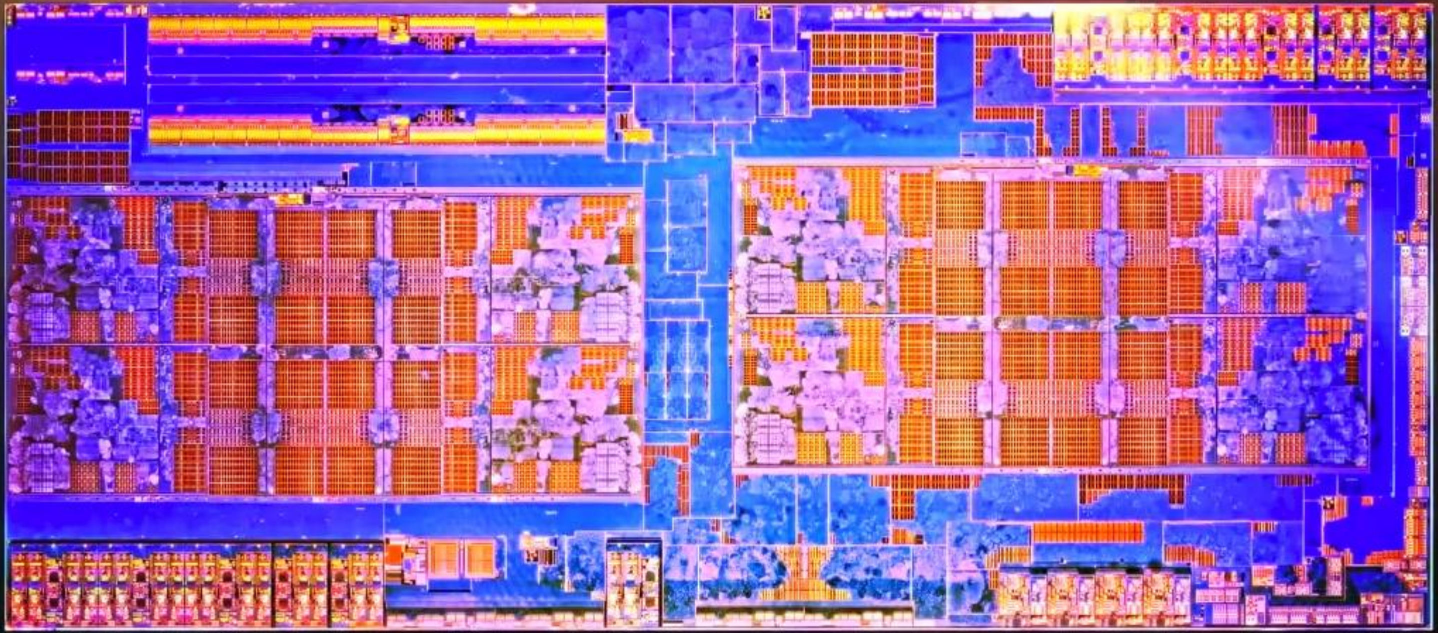
“A Plea for Lean Software” - Niklaus Wirth



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*

Efficiency and performance are **not dependant** on one another.

All those cores, idling...



Performance / Efficiency

C++ 17

Parallelize + Reduction

(map/reduce)

C++17 supports **parallel** versions of the STL *algorithms* (*many of them*)

=> WOW ! It became really simple to write parallel code 🎉

Eg.

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

```
template< class ExecutionPolicy, class ForwardIt, class T >
ForwardIt find( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, const T& value );
```

Not so fast ! Let’s see...

Parallel STL Algorithms

C++ 17

`ExecutionPolicy`

- `std::execution::seq`
  - same as non-parallel algorithm (invocations of element access functions are indeterminately ***sequenced*** in the calling thread)
- `std::execution::par`
  - execution may be ***parallelized*** (invocations of element access functions are permitted to execute in either the *invoking thread* or in a *thread created* by STL implicitly)
  - invocations executing in the same thread are ***indeterminately*** sequenced with respect to each other
- `std::execution::par_unseq`
  - execution may be ***parallelized***, ***vectorized***, or ***migrated*** across threads (by STL)
  - invocations of element access functions are permitted to execute:
    - in an **unordered** fashion
    - in *unspecified* threads
    - **unsequenced** with respect to one another, within each thread

C++

Powerful as hell. Can actually do anything.

God save you if something goes wrong.

Parallel STL Algorithms

```
template<class Iterator>
size_t seq_calc_sum(Iterator begin, Iterator end)
{
  size_t x = 0;
  std::for_each(begin, end, [&](int item) {
    x += item;
  });
  return x;
}
```

sequential (single thread)

Parallel STL Algorithms

C++ 17

```
template<class Iterator>
size_t par_calc_sum(Iterator begin, Iterator end)
{
  size_t x = 0;
  std::for_each(std::execution::par, begin, end, [&](int item) {
    x += item;     <= data race; fast, but often causes wrong result!
  });
  return x;
}
```

Parallel STL Algorithms

C++ 17

```
template<class Iterator>
size_t par_calc_sum(Iterator begin, Iterator end)
{
  size_t x = 0;
  std::mutex m;
  std::for_each(std::execution::par, begin, end, [&](int item) {
    std::lock_guard<std::mutex> guard(m);   <= ~90x slower than sequential version
    x += item;
  });
  return x;
}
```

Parallel STL Algorithms

C++ 17

```
template<class Iterator>
size_t par_calc_sum(Iterator begin, Iterator end)
{
  std::atomic<size_t> x = 0;
  std::for_each(std::execution::par, begin, end, [&](int item) {
    x += item; // or x.fetch_add(item);     <= ~50x slower than sequential version
  });
  return x;
}
```

Parallel STL Algorithms

Always Benchmark !

Don’t trust your instinct

Results

Box	non-parallelized	std::execution::par with std::mutex	std::execution::par with std::atomic
#1 (4 physical, 8 logical cores)	470+-4us	41200+-900us (90x slower, 600x+ less power-efficient)	23400+-140us (50x slower, 300x+ less power-efficient)
#2 (2 physical, 4 logical cores)	900+-150us	52500+-6000us (60x slower, 200x+ less power-efficient)	25100+-4500us (30x slower, 100x+ less power-efficient)

Parallel STL Algorithms

C++ 17

```
template<class RandomAccessIterator>
size_t par_calc_sum(RandomAccessIterator begin, RandomAccessIterator end)
{
  constexpr int NCHUNKS = 128;   // reduce the synchronization overhead by partitioning the load in chunks
  assert( (end-begin) % NCHUNKS == 0 );              // for simplicity of slide code
  const size_t szChunk = (end - begin) / NCHUNKS;   // size of a chunk

  RandomAccessIterator starts[NCHUNKS];              // compute start offsets for all chunks
  for (int i = 0; i < NCHUNKS; ++i) {
    starts[i] = begin + szChunk * i;
    assert(starts[i] < end);
  }

  std::atomic<size_t> total = 0;

  std::for_each(std::execution::par, starts, (starts + NCHUNKS), [&](RandomAccessIterator pos)
  {
    size_t partial_sum = 0;
    for (auto it = pos; it < pos + szChunk; ++it)
      partial_sum += *it; // NO synchronization (COLD)

    total += partial_sum; // synchronization (HOT)
  });

  return total;
}
```

Almost 2x FASTER than sequential version 👍

(on 8 core CPU)

Parallel STL Algorithms

C++ 17

std::reduce()

```
template<class Iterator>
size_t par_calc_sum(Iterator begin, Iterator end)
{
  return std::reduce(std::execution::par, begin, end, (size_t)0);
}
```

+

std::reduce() – just like our *partial sums* code, exploits the fact that *operation* which is used for reduce (default is: +) is **associative**.

```
template<class ExecutionPolicy, class ForwardIt, class T, class BinaryOp>
T reduce(ExecutionPolicy && policy, ForwardIt first, ForwardIt last, T init, BinaryOp binary_op);
```

~3% faster than our manual implementation 👍

(on 8 core CPU)

https://en.cppreference.com/w/cpp/algorithm/reduce

Parallel STL Algorithms

C++ 17

TL;DR: `std::reduce()` ⚡ rulezz !

Pretty much all other ***parallel*** algorithms are *difficult* to use properly:

- safe (no data races)
- with good performance results

  (on traditional architectures; exception NUMA/GPGPU)
- don't trust your instinct: Always Benchmark !

“Show me mooooooore code”

Let’s explore some real-world examples…
(cherry-picked from our codebase 🤫)

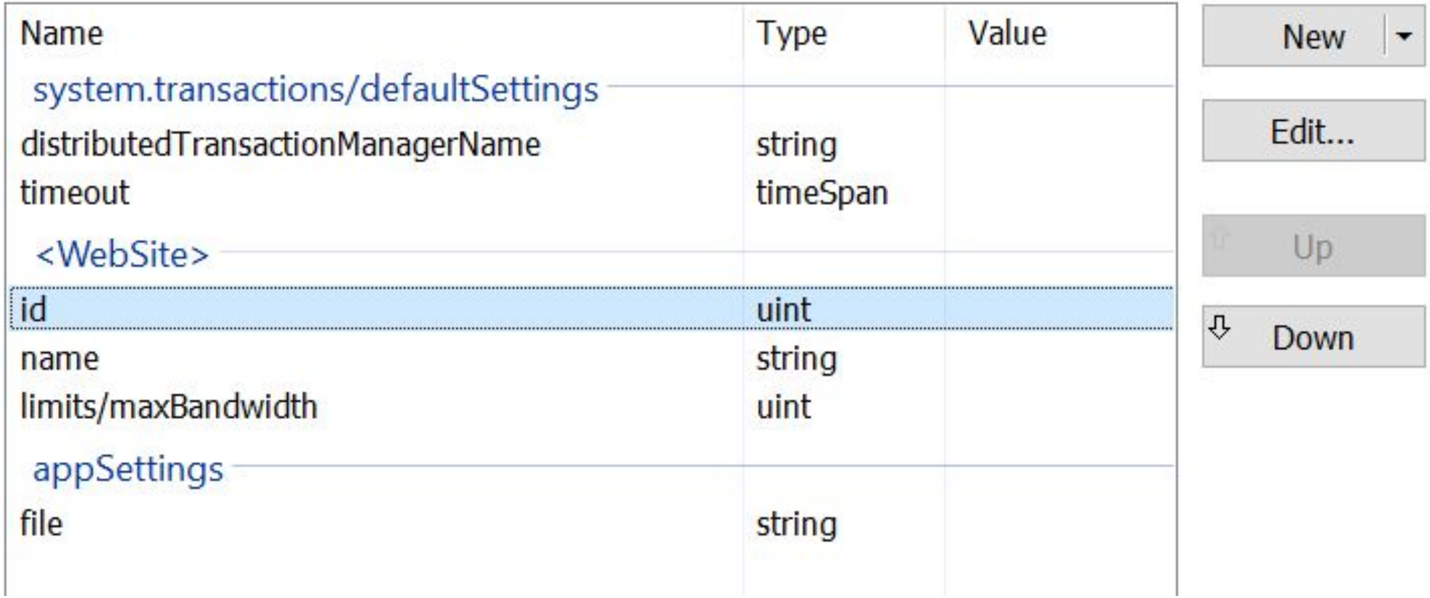
Calculating total number of unread messages.

```
// Raw loop version.  See anything wrong?
int MessagePool::CountUnreadMessages()  const
{
  int unreadCount = 0;

  for (size_t i = 0; i < mReaders.size(); ++i)
  {
      const vector<MessageItem *> & readMessages = Readers[i]->GetMessages();

      for (size_t j = 0; j < readMessages.size(); ++i)
      {
        if ( ! readMessages[j]->mRead )
          unreadCount++;
      }
  }
  return unreadCount;
}
```

Calculating total number of unread messages.

```
// Modern C++, with STL:
int MessagePool::CountUnreadMessages()  const
{
  return std::accumulate(
    begin(mReaders), end(mReaders),  0,
    [](int count, auto & reader)
    {
      const auto & readMessages = reader->GetMessages();

      return count + std::count_if( begin(readMessages),
                                    end(readMessages),
                                    []( const auto & message)
                                    {
                                       return ! message->mRead;
                                    });
    });
}
```

Enabling move operation (up/down) for a List item in user interface



Enabling move operation (up/down) for a List item in user interface

```
// Raw loop version.  See anything wrong?
bool CanListItemBeMoved(ListRow & aCurrentRow, bool aMoveUp)  const
{
  int min, max;
  vector<ListRow *> existingProperties = GetListRows(aCurrentRow.GetGroup());

  for (int i = 0; i < existingProperties.size(); ++i)
  {
      const int currentOrderNumber = existingProperties[i]->GetOrderNumber();
      if (currentOrderNumber < min)
          min = currentOrderNumber;
      if (currentOrderNumber > max)
          max = currentOrderNumber;
  }
  if (aMoveUp)
    return min < aCurrentRow.GetOrderNumber();
  else
    return max > aCurrentRow.GetOrderNumber();
}
```

C++ 14

Enabling move operation (up/down) for a List item in user interface

```
// Modern version, STL algorithm based
bool CanListItemBeMoved(ListRow & aCurrentRow,  bool aMoveUp) const
{
  vector<ListRow *> existingRows = GetListRows( aCurrentRow.GetGroup() );

  auto minmax = std::minmax_element(begin(existingRows),
                                    end(existingRows),
                                    []( auto & firstRow, auto & secondRow)
                                    {
                                      return firstRow.GetOrderNumber() <
                                             secondRow.GetOrderNumber();
                                    });
```

min

```
  if (aMoveUp)
    return (*minmax.first)->GetOrderNumber() < aCurrentRow.GetOrderNumber();
  else
    return (*minmax.second)->GetOrderNumber() > aCurrentRow.GetOrderNumber();
}
```

max

C++ 17

Enabling move operation (up/down) for a List item in user interface

```
// Modern version, STL algorithm based
bool CanListItemBeMoved(ListRow & aCurrentRow,  bool aMoveUp) const
{
  vector<ListRow *> existingRows = GetListRows( aCurrentRow.GetGroup() );

  auto [min, max] =  minmax_element(begin(existingRows),
                                    end(existingRows),
                                    []( auto & firstRow, auto & secondRow)
                                    {
                                      return firstRow.GetOrderNumber() <
                                             secondRow.GetOrderNumber();
                                    });

  if (aMoveUp)
    return min->GetOrderNumber() < aCurrentRow.GetOrderNumber();
  else
    return max->GetOrderNumber() > aCurrentRow.GetOrderNumber();
}
```

⇧ *structured binding*

Selecting attributes from XML nodes:

```
vector<XmlDomNode> childrenVector = parentNode.GetChildren();

set<string> childrenNames;
std::transform(begin(childrenVector), end(childrenVector),
               inserter(childrenNames, begin(childrenNames)),
                        getNodeNameLambda);


// A good, range based for, alternative:

for (auto & childNode : childrenVector)
    childrenNames.insert(getNodeNameLambda(childNode));


// Raw loop, see anything wrong?

for (unsigned int i = childrenVector.size(); i >= 0; --i)
  childrenNames.insert(getNodeNameLambda(childrenVector[i]));
```

Demo: FUN WITH STL

Server Nodes

Demo

Server Nodes

We have a <u>huge</u> network of server nodes.
Each server node contains a copy of a particular ***data*** `Value` (not necessarily unique).

`class Value` is a ***Regular*** type.

{ Assignable + Constructible + EqualityComparable + LessThanComparable }

The network is constructed in such a way that the nodes are ***sorted ascending*** with respect to their `Value` but their sequence might be **rotated** (left) by some offset.

Eg.

For the ***ordered*** node values:
`{ A, B, C, D, E, F, G, H }`

The **actual network** configuration might look like:
`{ D, E, F, G, H, A, B, C }`

Demo

Server Nodes

The network exposes the following APIs:

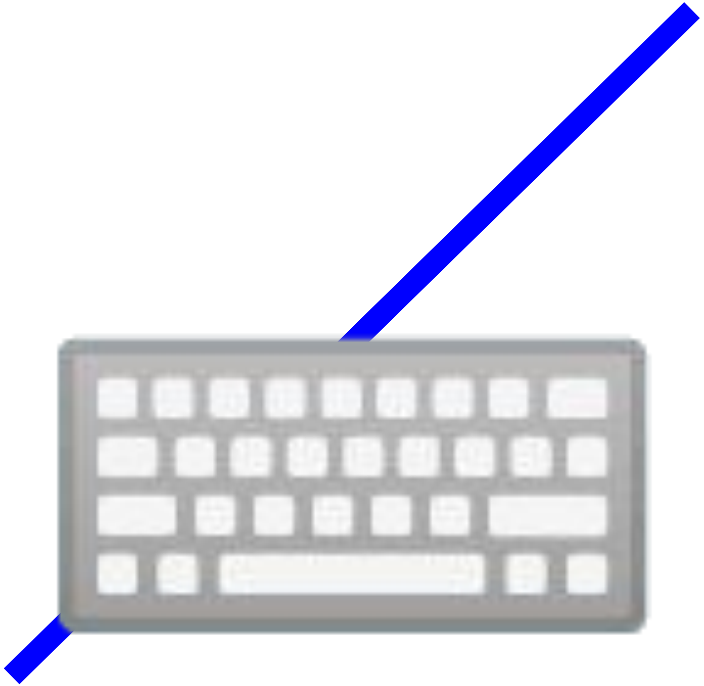
```
// gives the total number of nodes -  O(1)
size_t Count() const;

// retrieves the data from a given node -  O(1)
const Value & GetData(size_t index) const;

// iterator interface for the network nodes
vector<Value>::const_iterator BeginNodes() const;
vector<Value>::const_iterator EndNodes() const;
```

👉Implement a new API for the network, that efficiently finds a server node (address) containing a given data `Value`.

```
size_t GetNode(const Value & data) const
{
  // implement this
}
```

http://quick-bench.com

Demo: Server Nodes



// FIN