STL Algorithms
Principles and Practice

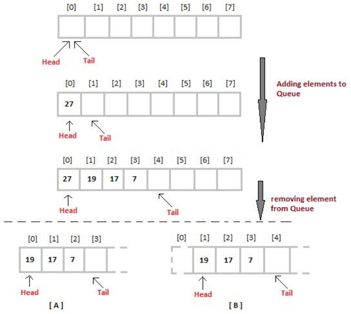
Victor Ciura - Technical Lead
Gabriel Diaconița - Senior Software Developer

February 2019

Agenda

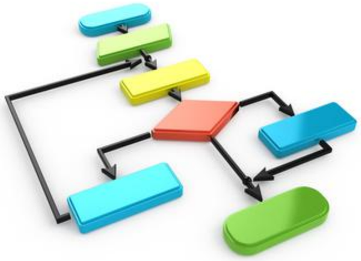
Part 0: STL Background

Part 1: Containers and Iterators



Part 2: STL Function Objects and Utilities

Part 3-4: STL Algorithms Principles and Practice



STL Background

(recap prerequisites)

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**

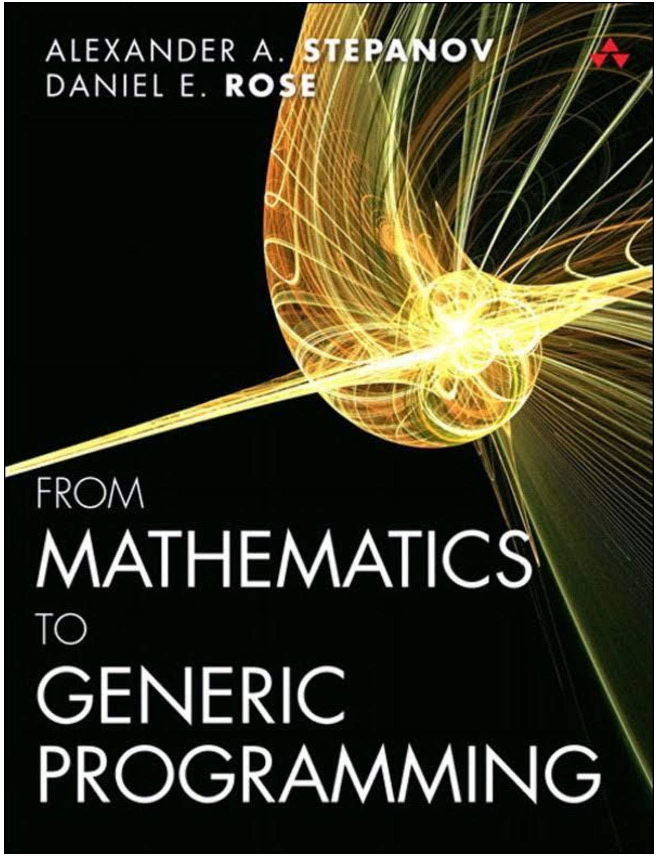
Alexander Stepanov (2002),

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"]

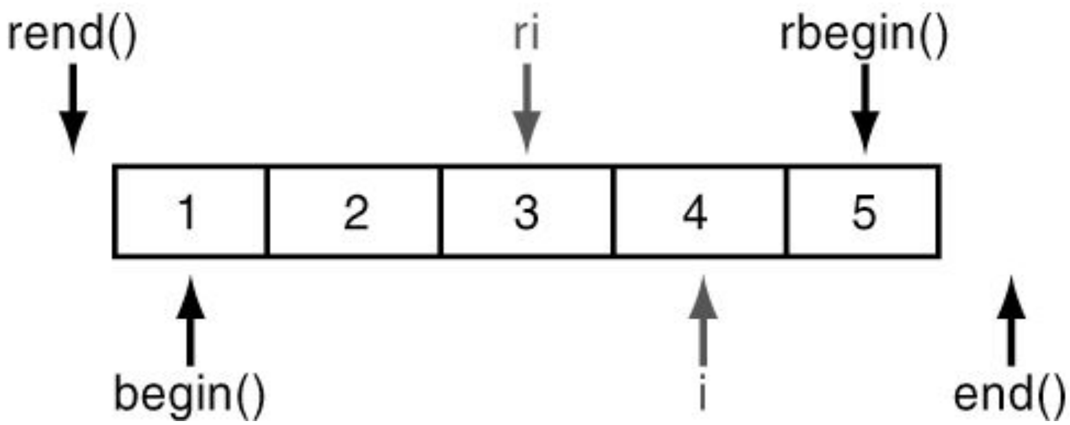
STL Iterators

- **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*

📎 Refactor existing code so that is prints numbers in reverse order.

The C way

```
vector<int> nrs = { 1, 549, 3, 52, 6 };
for (unsigned int n = 0; n < nrs.size(); ++n)
  cout << nrs[n] << " ";
```

Output: 1 549 3 52 6

```
vector<int> nrs = { 1, 549, 3, 52, 6 };
for (unsigned int i= nrs.size(); i>= 0; ++i)
  cout << nrs[n] << " ";
```

Output: ???

Code will execute forever! We just need the decrement operator ...or do we?

Old code forgotten during refactoring. Compiler will catch this

Can you spot any issues with this code?

C-style iteration vs *STL Iterators*

Refactor existing code so that is prints numbers in reverse order.

The `STL Iterators` way

```
vector<int> nrs = { 1, 549, 3, 52, 6 };
for (auto i = nrs.begin(), endIt = nrs.end(); i != endIt; ++i)
  cout << *it << " ";
```

Output: 1 549 3 52 6

```
vector<int> nrs = { 1, 549, 3, 52, 6 };
for (auto it = nrs.rbegin(), endIt = nrs.rend(); i != endIt; ++it)
  cout << *it << " ";
```

Output: 6 52 3 549 1

Can you spot any issues with this code?

Old code forgotten during refactoring.
Compiler will catch this

C-style iteration vs STL Iterators

📎 Refactor existing code so that is prints numbers in reverse order.

The `range-for` way

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto i : numbers)
  cout << i << " ";
```

Output: 1 549 3 52 6

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto i : reverse(numbers))
  cout << i << " ";
```

Output: 6 52 3 549 1

✓ 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; }
```

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

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

for ( auto & key : IterateFirst(m) )  { cout << key << endl; }

for ( auto & val : IterateSecond(m) ) { cout << val << endl; }
```

Using the same technique shown for `reverse()` iteration adaptor, implement `IterateFirst()` and `IterateSecond()` adaptors.

Email solutions to: gabriel.diaconita@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());
```

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; });
```

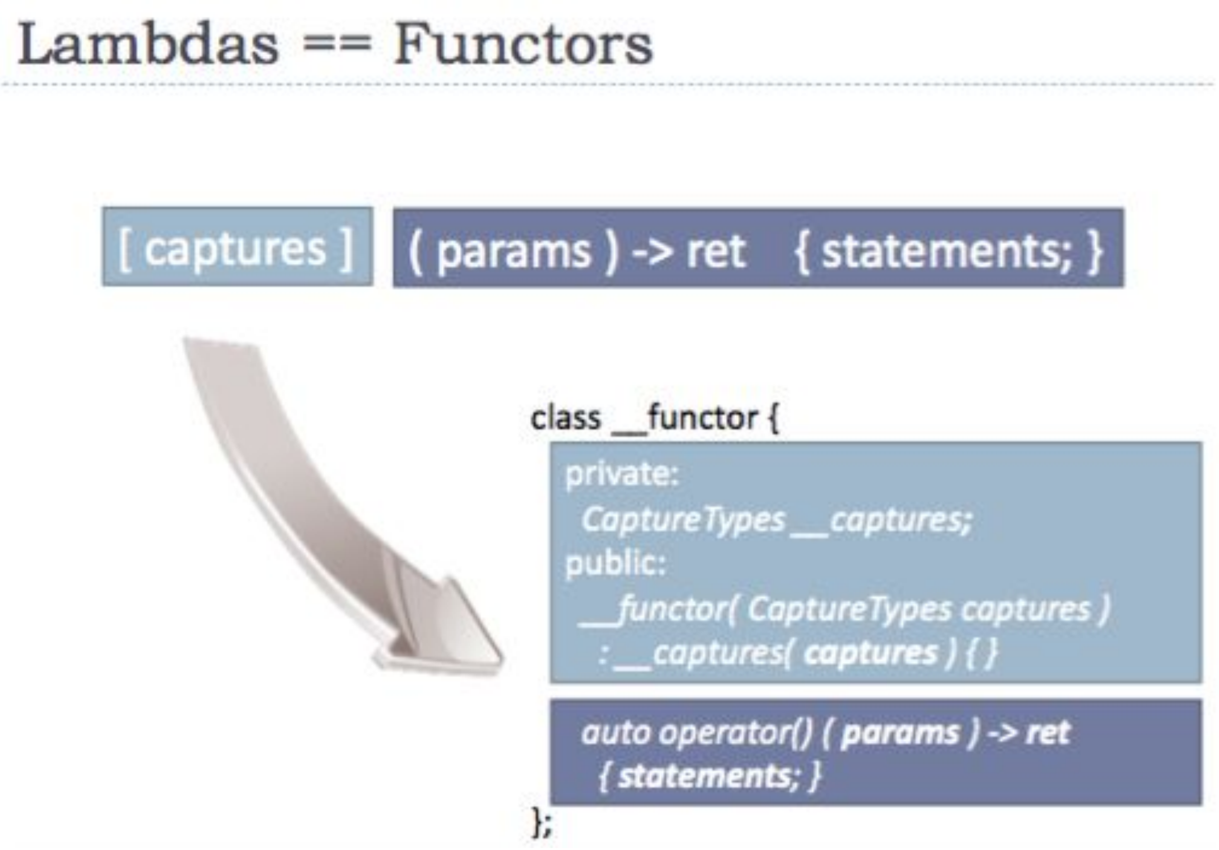
Lambda Functions

```
[ 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 :

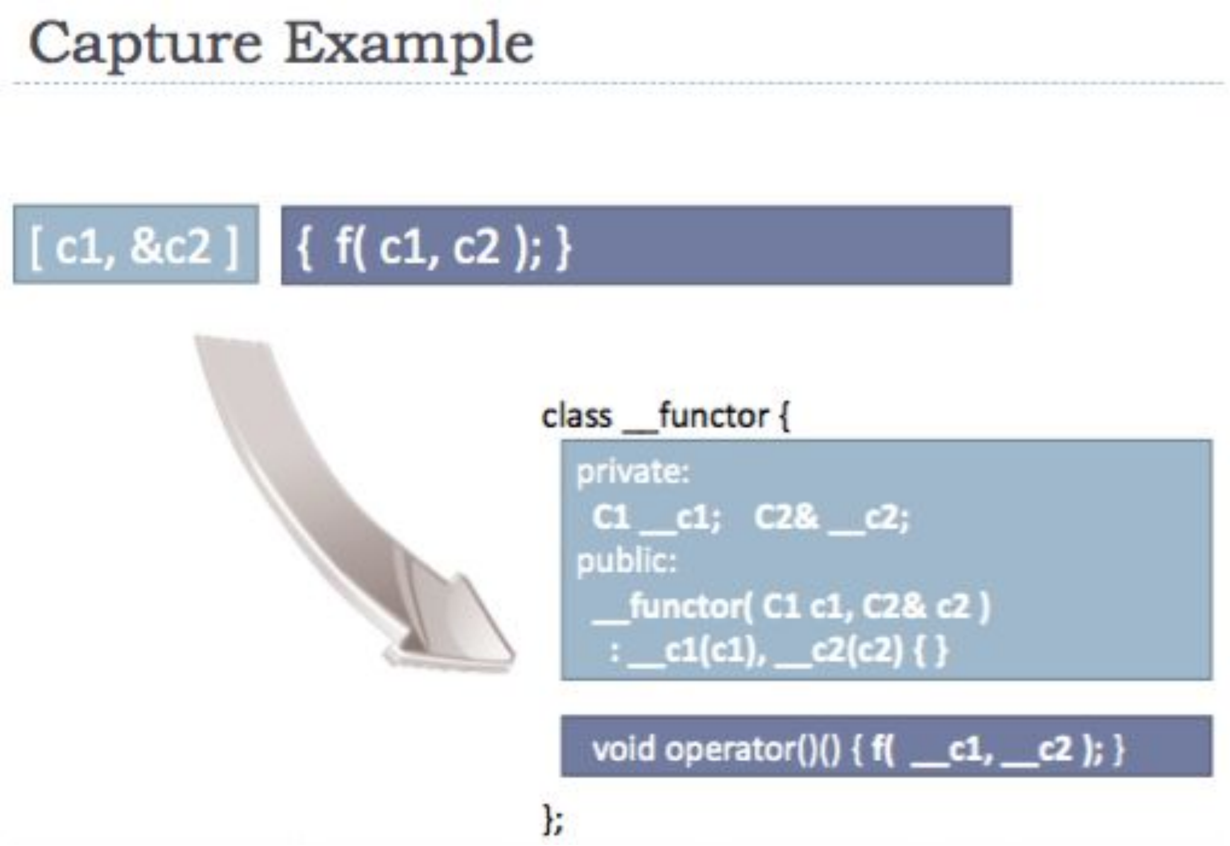
- **[a, &b]** where **a** is captured by ***value*** and **b** is captured by ***reference***.
- **[this]** captures the **this** pointer by ***value***
- **[&]** captures all automatic variables **used** in the body of the lambda by ***reference***
- **[=]** captures all automatic variables **used**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

Anatomy of A Lambda



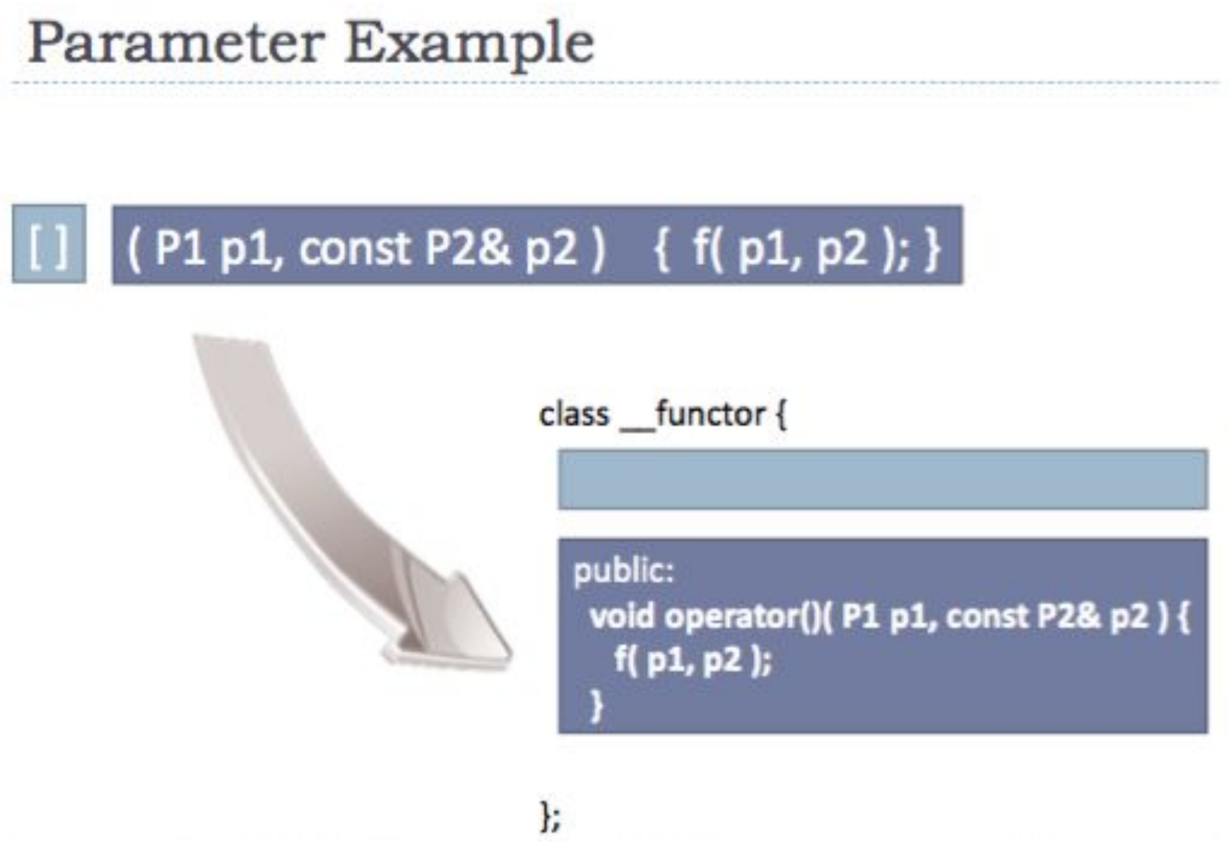
credit: Herb Sutter - "*Lambdas, Lambdas Everywhere*"
https://www.youtube.com/watch?v=rcgRY7sOA58

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; } );

// 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; });
```

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 ?

- We can’t have simplicity **everywhere**
- The problems we’re trying to solve or model are **complicated**
- Moving complexity to a **library** (or another **abstraction**) is good
- Complicated **guidelines** that lead us to writing simpler code are good
    - Being <u>forced to think</u> about resources, lifetime management, invariants, etc. is also good, even if it’s sometimes painful.

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
- Not about skipping or leaving out
  - error handling
  - testing
  - documentation
  - meaningful names

Kate Gregory, *"It's Complicated"*, Meeting C++ 2017

Why prefer to use (STL) algorithms?

Modern C++ (C++11/14/17 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

- 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**

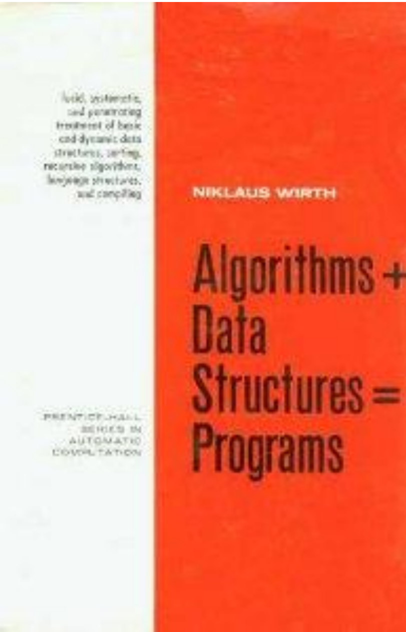
  (auto-vectorization, auto-parallelization, loop unrolling, dependency checking, etc.).

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 !

Performance / Efficiency

Parallelize + Reduction

(map/reduce)

C++17 supports parallel versions of the std::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

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;
}
```

Parallel STL Algorithms

```
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

```
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

```
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

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

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

  std::atomic<size_t> total = 0;

  std::for_each(std::execution::par, starts, starts + NCHUNKS, [&](RandomAccessIterator s)
  {
    size_t partial_sum = 0;
    for (auto it = s; it < s + sz; ++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 **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 !

Homework

Solve these two Advent of Code challenges, using constructs presented in this course (STL data structures, algorithms, lambda functions, range-for, etc):

https://adventofcode.com/2018/day/9 **EASY**

https://adventofcode.com/2018/day/13 **MEDIUM**

Email solutions to gabriel.diaconita@caphyon.com

See you in 2 weeks…

Don’t forget about your assignments

Homework