STL Algorithms
Principles and Practice

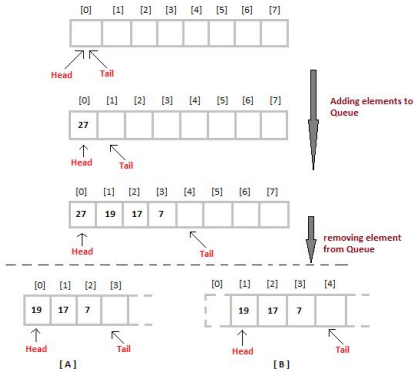
Victor Ciura - Technical Lead

Gabriel Diaconița - Senior Software Developer

February 2020

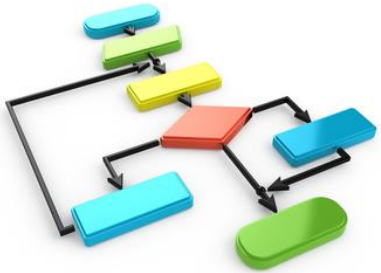
Agenda

Part 1: Containers and Iterators



Part 2: STL Function Objects and Utilities

Parts 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**

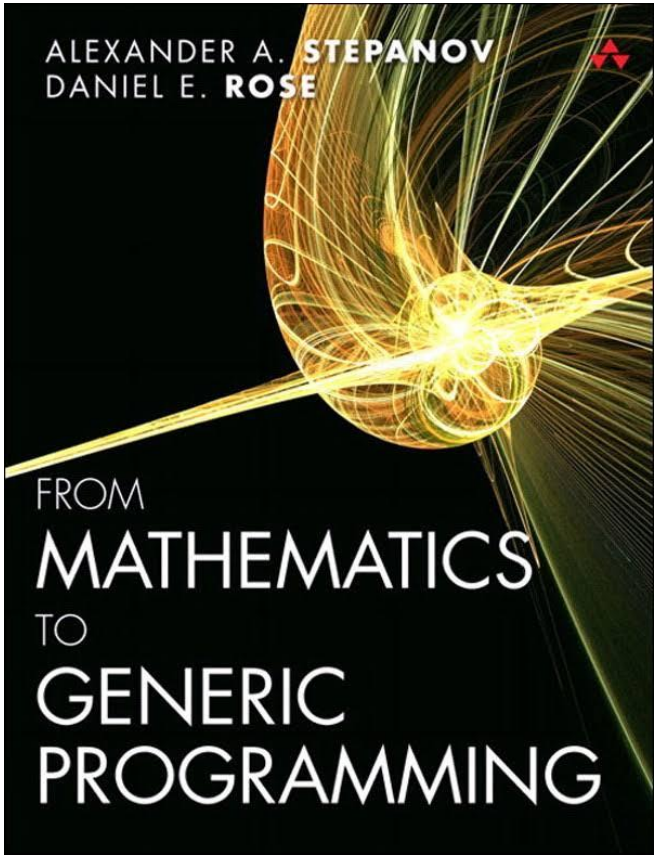
Alexander Stepanov (2002),

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"

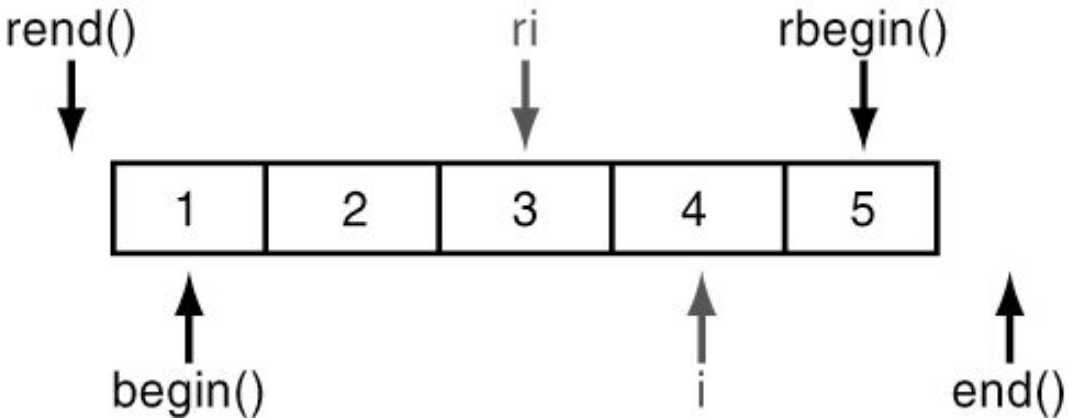
STL Iterators

- **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*

📋 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] << " ";
```

OUTPUT

S T L

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

???

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

Can you spot any issues with this code?

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] << " ";
```

OUTPUT

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 << " ";
```

OUTPUT

S T L

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

S T L

Can you spot any issues with this code?

Old code forgotten during refactoring.
Induction variable has different name

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`


```
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:` `pca@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

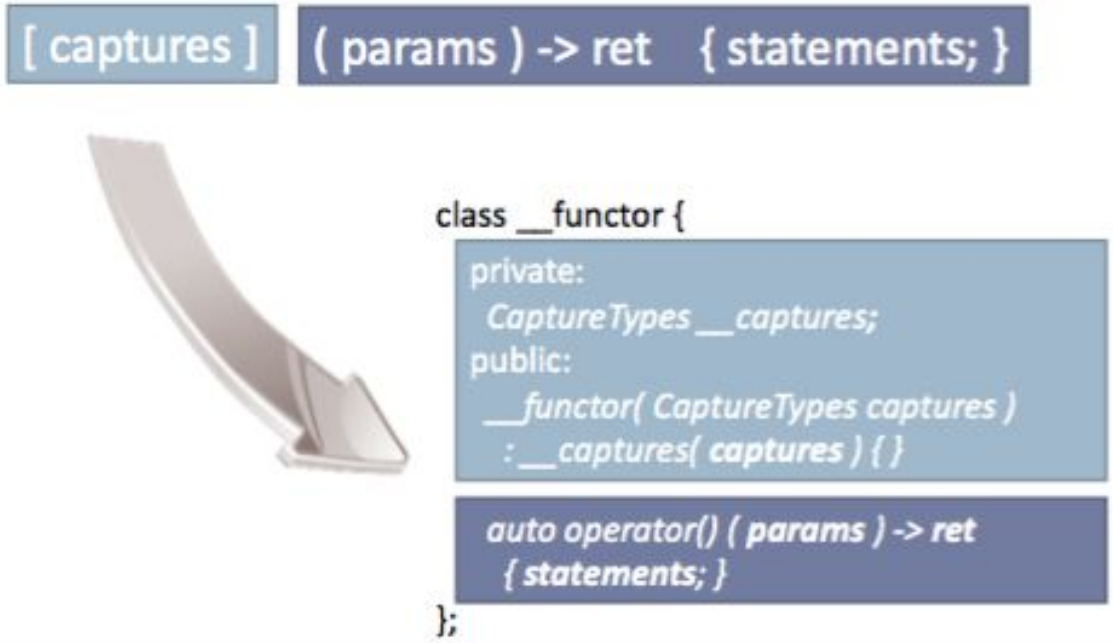
```
[ capture-list ] ( params ) mutable(optional) -> ret { body }

[ 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

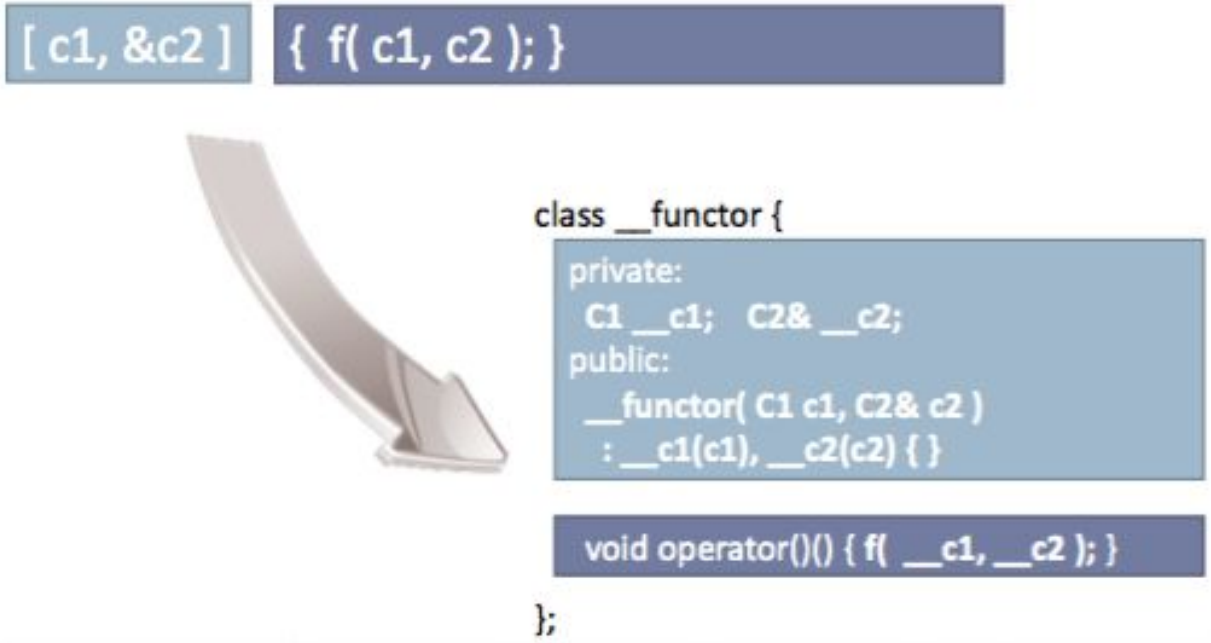
Lambdas == Functors



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

Anatomy of A Lambda

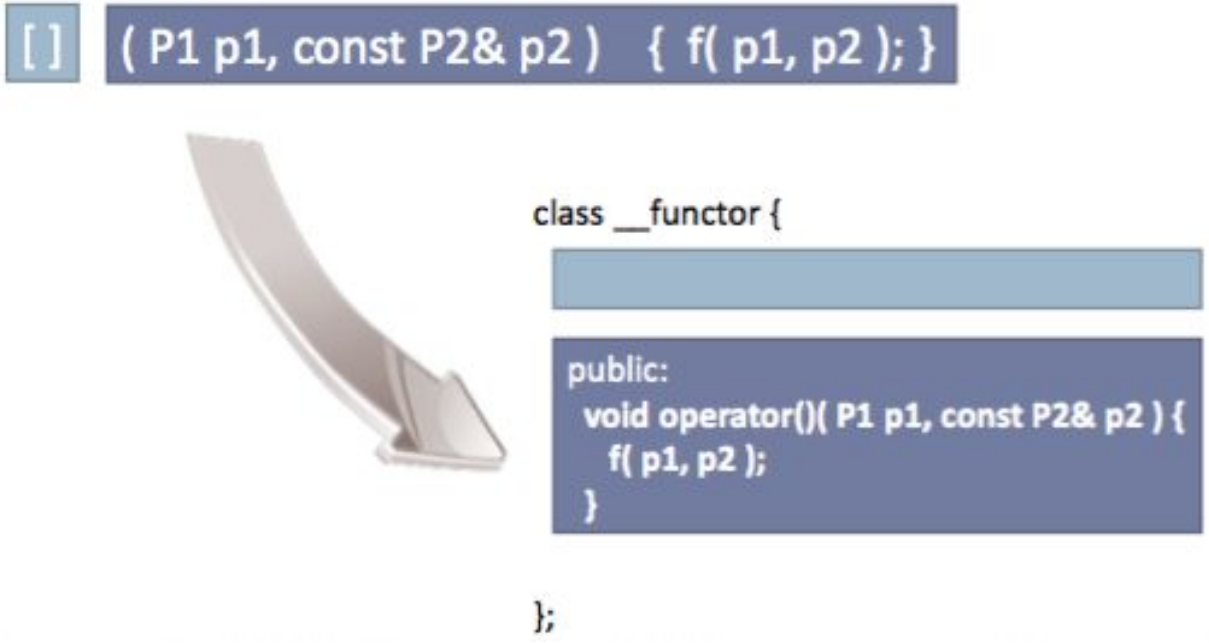
Capture Example



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

Anatomy of A Lambda

Parameter Example



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
- Not 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

- 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.).

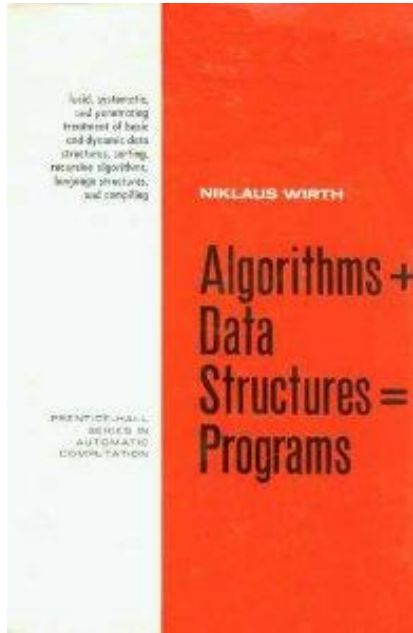
The difference between **Efficiency** and **Performance**

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



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

Performance / Efficiency

Parallelize + Reduction

(map/reduce)

C++ 17

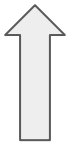
C++17 supports parallel versions of the std::algorithms (*many of them*)

=> 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

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)
{
  // 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

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 !

{ Practical Code

Practical Code

Source: Advent of Code 2019, day 22

Given a deck of playing cards, implement the following operations:

1. `DealNew()` Take all cards (from *front*) of deck and insert them into the *front* of a new card deck

2. `Cut(int N)` `N > 0`: Take *N* cards from the *deck front* and insert them to the *deck back*
   `N < 0`: Take *N* cards from the *deck back* and insert them to the *deck front*

Practical Code

`DealNew()` Take all cards (from *front*) of deck and insert them into the *front* of a new card deck

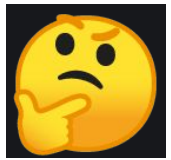
```
vector<Card> DealNew(const vector<Card>& deck)
{
  vector<Card> newDeck = deck;

  for (int i = 0; i < deck.size(); ++i)
    newDeck[deck.size() - i - 1] = deck[i];

  return newDeck;
}
```

We can do better

Practical Code

`DealNew()` Take all cards (from *front*) of deck and insert them into the *front* of a new card deck

```
vector<Card> DealNew(const vector<Card>& deck)
{
  vector<Card> newDeck;
  for (auto & c : deck)
    newDeck.insert(begin(newDeck), c);

  return newDeck;
}
```



We can do better still

Practical Code

`DealNew()` Take all cards (from *front*) of deck and insert them into the *front* of a new card deck

```
vector<Card> DealNew(const vector<Card>& deck)
{
  vector<Card> newDeck = deck;

  reverse(begin(newDeck), end(newDeck));

  return newDeck;
}
```

Practical Code

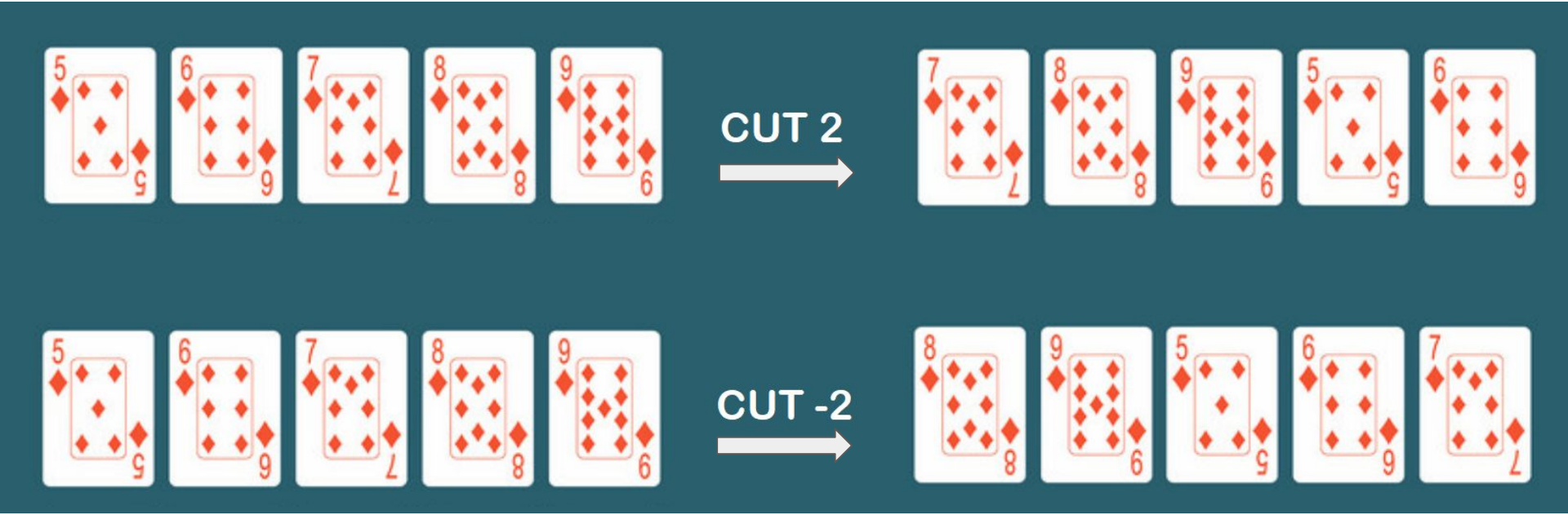
```
reverse(begin(newDeck), end(newDeck));
```

- Easy to read. A verb
- Does what it says
- Not fiddling around with indices
- A fast, tested and proven STL algorithm
- Bonus: Shorter

Practical Code

`Cut(int N)` `N > 0`: Take *N* cards from the *deck front* and insert them to the *deck back*

`N < 0`: Take *N* cards from the *deck back* and insert them to the *deck front*



Practical Code

`Cut(int N)` `N > 0`: Take *N* cards from the *deck front* and insert them to the *deck back*

`N < 0`: Take *N* cards from the *deck back* and insert them to the *deck front*

```
vector<Card> Cut(const vector<Card> & v, int n)
{

// ...

}
```

Practical Code

`Cut(int N)` `N > 0`: Take *N* cards from the *deck front* and insert them to the *deck back*

`N < 0`: Take *N* cards from the *deck back* and insert them to the *deck front*

```
vector<Card> Cut(const vector<Card> & deck, int n)
{
  vector<Card> cutDeck(deck.size());

  if (n > 0)
  {
    for (int i = 0; i < n; ++i)
      cutDeck[v.size() - n + i] = deck[i];
    for (int i = 0; i < deck.size() - n; ++i)
      cutDeck[i] = deck[i + n];

 } else
 {
   n = abs(n);
   for (int i = 0; i < n; ++i)
     cutDeck[i] = deck[v.size() - n + i];
   for (int i = n; i < deck.size(); ++i)
     cutDeck[i] = deck[i - n];
 }

  return cutDeck;
}
```

We can do better

Practical Code

`Cut(int N)` `N > 0`: Take *N* cards from the *deck front* and insert them to the *deck back*

`N < 0`: Take *N* cards from the *deck back* and insert them to the *deck front*

```
vector<Card> Cut(const vector<Card> & deck, int n)
{
  auto cutDeck = deck;

  if (n > 0)
    rotate(cutDeck.begin(), cutDeck.begin() + n, cutDeck.end());
  else
    rotate(cutDeck .rbegin(), cutDeck.rbegin() - n, cutDeck.rend());

  return cutDeck;
}
```

Practical Code

```
for (int i = 0; i < n; ++i)
  cutDeck[v.size() - n + i] = deck[i];
for (int i = 0; i < deck.size() - n; ++i)
  cutDeck[i] = deck[i + n];
```

VS

```
rotate(cutDeck.begin(),
       cutDeck.begin() + n,
       cutDeck.end());
```

- Easy to read. A verb
- Fast to write without errors
- Does what it says
- Not fiddling around with indices
- A fast, tested and proven STL algorithm
- It can work in-place on collections
- Bonus: Shorter

Practical Code }

Homework

Comparative Reviews

You have a large collection of proposals for a conference.

As the organizer, you are tasked with devising an **evaluation system** for submissions.

Your strategy: ask all reviewers to rank proposals against each other.

Email solutions to pca@caphyon.com

Homework

Comparative Reviews

Devise a program that will repeatedly show **3 randomly chosen** proposals to the reviewer, asking them to rank comparatively: 1, 2, 3 | 1, 3, 2 | 2, 1, 3 | etc

This cycle runs forever (continuously showing 3 random proposals), until the reviewer is satisfied that they ranked enough submissions.

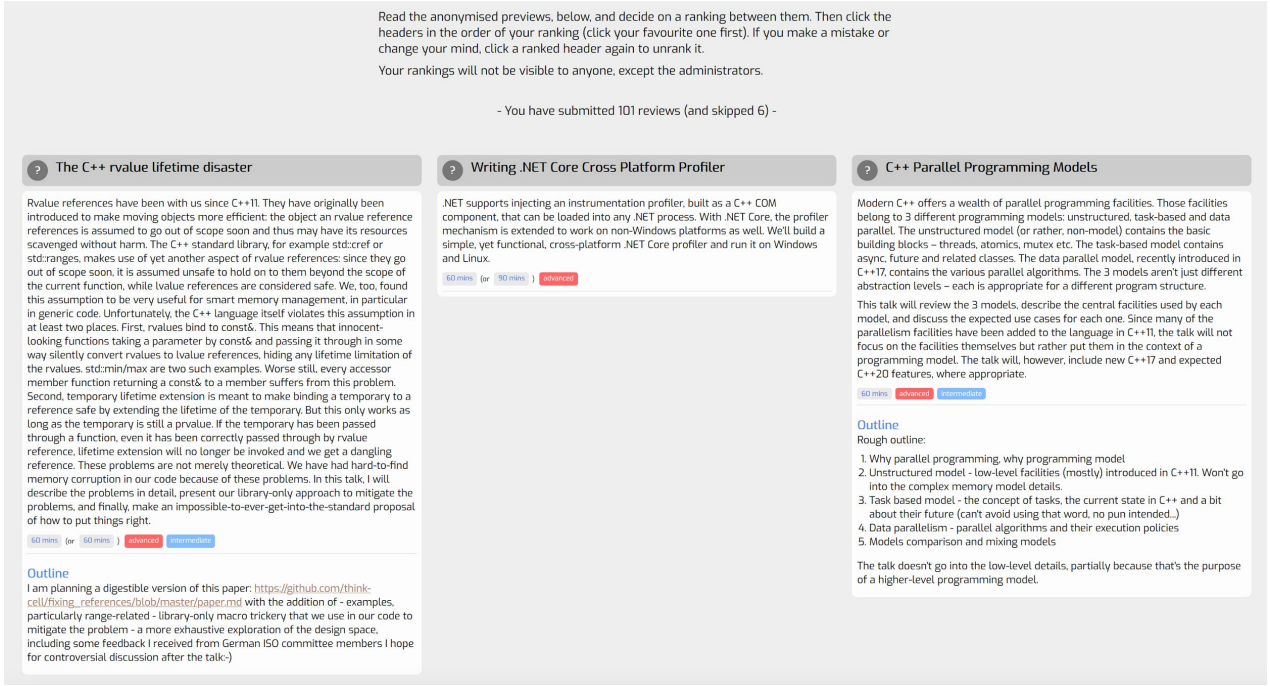
Note that proposals may show up multiple times, in different combinations with each other and must be ranked in that context.

Compute the final ranking of ALL proposals (eg. Top 100) based on the votes from all reviewers, after they all finished the evaluation.

Email solutions to pca@caphyon.com

Homework

Comparative Reviews



Read the anonymised previews, below, and decide on a ranking between them. Then click the headers in the order of your ranking (click your favourite one first). If you make a mistake or change your mind, click a ranked header again to unrank it.

Your rankings will not be visible to anyone, except the administrators.

- You have submitted 101 reviews (and skipped 6) -

? The C++ rvalue lifetime disaster

Rvalue references have been with us since C++11. They have originally been introduced to make moving objects more efficient: the object an rvalue reference references is assumed to go out of scope soon and thus may have its resources scavenged without harm. The C++ standard library, for example std::cref or std::ranges, makes use of yet another aspect of rvalue references: since they go out of scope soon, it is assumed unsafe to hold on to them beyond the scope of the current function, while lvalue references are considered safe. We, too, found this assumption to be very useful for smart memory management, in particular in generic code. Unfortunately, the C++ language itself violates this assumption in at least two places. First, rvalues bind to const&. This means that innocent-looking functions taking a parameter by const& and passing it through in some way silently convert rvalues to lvalue references, hiding any lifetime limitation of the rvalues. std::min/max are two such examples. Worse still, every accessor member function returning a const& to a member suffers from this problem. Second, temporary lifetime extension is meant to make binding a temporary to a reference safe by extending the lifetime of the temporary. But this only works as long as the temporary is still a prvalue. If the temporary has been passed through a function, even it has been correctly passed through by rvalue reference, lifetime extension will no longer be invoked and we get a dangling reference. These problems are not merely theoretical. We have had hard-to-find memory corruption in our code because of these problems. In this talk, I will describe the problems in detail, present our library-only approach to mitigate the problems, and finally, make an impossible-to-ever-get-into-the-standard proposal of how to put things right.

60 mins (or 60 mins) advanced intermediate

Outline

I am planning a digestible version of this paper: https://github.com/think-cell/fixing_references/blob/master/paper.md with the addition of - examples, particularly range-related - library-only macro trickery that we use in our code to mitigate the problem - a more exhaustive exploration of the design space, including some feedback I received from German ISO committee members I hope for controversial discussion after the talk:-)

? Writing .NET Core Cross Platform Profiler

.NET supports injecting an instrumentation profiler, built as a C++ COM component, that can be loaded into any .NET process. With .NET Core, the profiler mechanism is extended to work on non-Windows platforms as well. We'll build a simple, yet functional, cross-platform .NET Core profiler and run it on Windows and Linux.

60 mins (or 90 mins) advanced

? C++ Parallel Programming Models

Modern C++ offers a wealth of parallel programming facilities. Those facilities belong to 3 different programming models: unstructured, task-based and data parallel. The unstructured model (or rather, non-model) contains the basic building blocks – threads, atomics, mutex etc. The task-based model contains async, future and related classes. The data parallel model, recently introduced in C++17, contains the various parallel algorithms. The 3 models aren't just different abstraction levels – each is appropriate for a different program structure.

This talk will review the 3 models, describe the central facilities used by each model, and discuss the expected use cases for each one. Since many of the parallelism facilities have been added to the language in C++11, the talk will not focus on the facilities themselves but rather put them in the context of a programming model. The talk will, however, include new C++17 and expected C++20 features, where appropriate.

60 mins advanced intermediate

Outline

Rough outline:

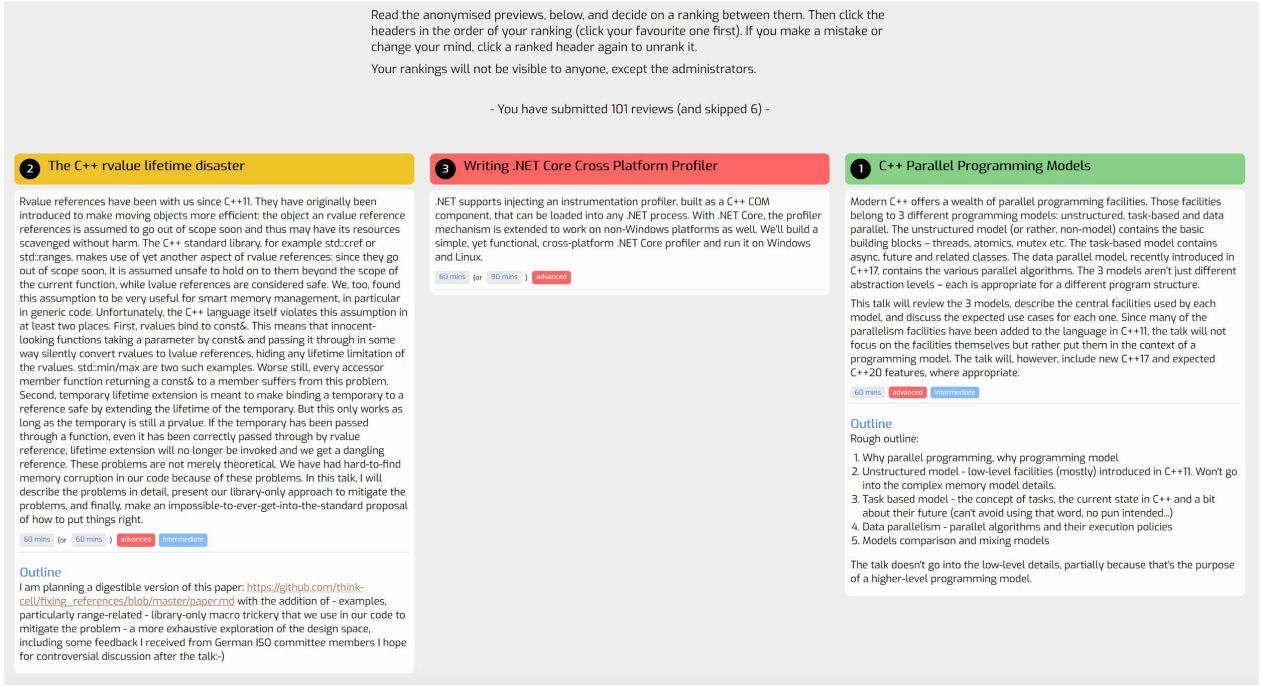
1. Why parallel programming, why programming model
2. Unstructured model - low-level facilities (mostly) introduced in C++11. Won't go into the complex memory model details.
3. Task based model - the concept of tasks, the current state in C++ and a bit about their future (can't avoid using that word, no pun intended...)
4. Data parallelism - parallel algorithms and their execution policies
5. Models comparison and mixing models

The talk doesn't go into the low-level details, partially because that's the purpose of a higher-level programming model.

Email solutions to pca@caphyon.com

Homework

Comparative Reviews



Read the anonymised previews, below, and decide on a ranking between them. Then click the headers in the order of your ranking (click your favourite one first). If you make a mistake or change your mind, click a ranked header again to unrank it.

Your rankings will not be visible to anyone, except the administrators.

- You have submitted 101 reviews (and skipped 6) -

2 The C++ rvalue lifetime disaster

Rvalue references have been with us since C++11. They have originally been introduced to make moving objects more efficient: the object an rvalue reference references is assumed to go out of scope soon and thus may have its resources scavenged without harm. The C++ standard library, for example std::cref or std::ranges, makes use of yet another aspect of rvalue references: since they go out of scope soon, it is assumed unsafe to hold on to them beyond the scope of the current function, while lvalue references are considered safe. We, too, found this assumption to be very useful for smart memory management, in particular in generic code. Unfortunately, the C++ language itself violates this assumption in at least two places. First, rvalues bind to const&. This means that innocent-looking functions taking a parameter by const& and passing it through in some way silently convert rvalues to lvalue references, hiding any lifetime limitation of the rvalues. std::min/max are two such examples. Worse still, every accessor member function returning a const& to a member suffers from this problem. Second, temporary lifetime extension is meant to make binding a temporary to a reference safe by extending the lifetime of the temporary. But this only works as long as the temporary is still a prvalue. If the temporary has been passed through a function, even it has been correctly passed through by rvalue reference, lifetime extension will no longer be invoked and we get a dangling reference. These problems are not merely theoretical. We have had hard-to-find memory corruption in our code because of these problems. In this talk, I will describe the problems in detail, present our library-only approach to mitigate the problems, and finally, make an impossible-to-ever-get-into-the-standard proposal of how to put things right.

60 mins (or 60 mins) advanced intermediate

Outline

I am planning a digestible version of this paper: https://github.com/think-cell/fixing_references/blob/master/paper.md with the addition of - examples, particularly range-related - library-only macro trickery that we use in our code to mitigate the problem - a more exhaustive exploration of the design space, including some feedback I received from German ISO committee members I hope for controversial discussion after the talk:-)

3 Writing .NET Core Cross Platform Profiler

.NET supports injecting an instrumentation profiler, built as a C++ COM component, that can be loaded into any .NET process. With .NET Core, the profiler mechanism is extended to work on non-Windows platforms as well. We'll build a simple, yet functional, cross-platform .NET Core profiler and run it on Windows and Linux.

60 mins (or 90 mins) advanced

1 C++ Parallel Programming Models

Modern C++ offers a wealth of parallel programming facilities. Those facilities belong to 3 different programming models: unstructured, task-based and data parallel. The unstructured model (or rather, non-model) contains the basic building blocks – threads, atomics, mutex etc. The task-based model contains async, future and related classes. The data parallel model, recently introduced in C++17, contains the various parallel algorithms. The 3 models aren't just different abstraction levels – each is appropriate for a different program structure.

This talk will review the 3 models, describe the central facilities used by each model, and discuss the expected use cases for each one. Since many of the parallelism facilities have been added to the language in C++11, the talk will not focus on the facilities themselves but rather put them in the context of a programming model. The talk will, however, include new C++17 and expected C++20 features, where appropriate.

60 mins advanced intermediate

Outline

Rough outline:

1. Why parallel programming, why programming model
2. Unstructured model - low-level facilities (mostly) introduced in C++11. Won't go into the complex memory model details.
3. Task based model - the concept of tasks, the current state in C++ and a bit about their future (can't avoid using that word, no pun intended...)
4. Data parallelism - parallel algorithms and their execution policies
5. Models comparison and mixing models

The talk doesn't go into the low-level details, partially because that's the purpose of a higher-level programming model.

Email solutions to pca@caphyon.com

Homework

Comparative Reviews

Get your conference proposal data from
https://cpponsea.uk/news/announcing-speakers-for-2020.html

The program **loop** will run for each of the reviewers, until they are each satisfied / tired with their ranking work

After the program is done executing, save the proposal ranking in `ranking.out` file

Email solutions to pca@caphyon.com

Homework

Comparative Reviews: Prototype

first set to rank

your KB input

second set to rank

your KB input

```
Ranking started. User: Gabriel
Rank with 123, 132, 213, 231, 312, 321

1. Adi Shavit - Coroutine X-Rays and Other Magical Superpowers
2. Dawid Zalewski - Structured bindings uncovered
3. Kevlin Henney - Lambda? You Keep Using that Letter

Rank: 132

1. Luna Kirkby  - Mind the Bear Traps!
2. Timur Doumler - Real-time STL
3. Adi Shavit - Coroutine X-Rays and Other Magical Superpowers

Rank: 312
```

Email solutions to pca@caphyon.com

Homework

Comparative Reviews: Prototype

proposals.txt

```
Adi Shavit - Coroutine X-Rays and Other Magical Superpowers
A.J. Orians - Improving Readability With Class Template Argument Deduction
Alexander Maslennikov - Algorithmic and microarchitecture optimizations of C++ applications
Anastasiia Kazakova - C++ ecosystem: the renaissance edition
Anders Schau Knatten - Just Enough Assembly for Compiler Explorer
Andrzej Warzyński - How compilers work: introduction to LLVM passes
Ansel Sermersheim and Barbara Geller - Refactoring Undefined Behavior using Any, Variant, and Optional from C+
Arnaud Desitter - Reducing Memory Allocations in a Large C++ Application
Arne Mertz - Phantastic Code Smells and Where to Find Them
Arno Schoedl - From Iterators To Ranges — The Upcoming Evolution Of the Standard Library
Björn Fahller - What Do You Mean by "Cache Friendly"?
Boguslaw Cyganek - How accurate we are? A refresher on the floating-point computations and the standard librar
Clare Macrae - Quickly Testing Legacy C++ Code with Approval Tests
Danila Kutenin - C++ STL best and worst performance features and how to learn from them
Dawid Zalewski - Structured bindings uncovered
Fergus Cooper - C++20: All the small things
Fred Tingaud - Clang-based Refactoring, or How to Refactor Millions of Line of Code Without Alienating your Co
Hendrik Niemeyer - An Introduction to C++20's Concepts
James Pascoe - Combining Modern C++ and Lua
JeanHeyd "ThePhD" Meneide - Burning Silicon: Speed for Transcoding in C++23
Jonathan Müller - Using C++20's Three-way Comparison <=>
Jon Kalb - Best Practices for Object-Oriented Programming
Juan Pedro Bolivar Puente - Squaring the circle: value-oriented design in an object-oriented system
Kate Gregory - Naming is Hard: Let's Do Better
Kevlin Henney - Lambda? You Keep Using that Letter
Luna Kirkby - Mind the Bear Traps!
Mateusz Pusz - Rethinking the Way We Do Templates in C++ even more
Matt Godbolt - Correct by Construction: APIs That Are Easy to Use and Hard to Misuse
Neil Horlock - No more secrets? Why your secrets aren't safe and what you can do about it.
Patrick Mintram - Debugging Concepts 101
Pavel Novikov - Serialization in C++ has never been easier! But wait, there's more...
Rainer Grimm - Concepts in C++20: A Evolution or a Revolution?
Sandor Dargo - Undefined behaviour in the STL
Shachar Langbeheim - Data-Oriented Design for Object-Oriented Programmers
Sy Brand - Live Compiler Development with Cross-Platform Tooling
Timur Doumler - Real-time STL
Tina Ulbrich and Niel Waldren - Pythonic C++
Tony Van Eerd - An Introduction to Lock-free Programming
Victor Ciura - Avoid Success at All Costs
Viet Le - Packaging and distributing C++ for fun and profit
Vittorio Romeo - C++11/14 at scale - what have we learned?
Yuri Minaev - Paranoid's take on C++ code review
```

ranking.out

```
Dawid Zalewski - Structured bindings uncovered
Hendrik Niemeyer - An Introduction to C++20's Concepts
Victor Ciura - Avoid Success at All Costs
JeanHeyd "ThePhD" Meneide - Burning Silicon: Speed for Transcoding in C++23
Jonathan Müller - Using C++20's Three-way Comparison <=>
Anders Schau Knatten - Just Enough Assembly for Compiler Explorer
Tony Van Eerd - An Introduction to Lock-free Programming
Björn Fahller - What Do You Mean by "Cache Friendly"?
Adi Shavit - Coroutine X-Rays and Other Magical Superpowers
Neil Horlock - No more secrets? Why your secrets aren't safe and what you can do about it.
Patrick Mintram - Debugging Concepts 101
Pavel Novikov - Serialization in C++ has never been easier! But wait, there's more...
Fergus Cooper - C++20: All the small things
Fred Tingaud - Clang-based Refactoring, or How to Refactor Millions of Line of Code Without Alienating your C
Ansel Sermersheim and Barbara Geller - Refactoring Undefined Behavior using Any, Variant, and Optional from C
Arnaud Desitter - Reducing Memory Allocations in a Large C++ Application
Arne Mertz - Phantastic Code Smells and Where to Find Them
James Pascoe - Combining Modern C++ and Lua
Alexander Maslennikov - Algorithmic and microarchitecture optimizations of C++ applications
Shachar Langbeheim - Data-Oriented Design for Object-Oriented Programmers
Jon Kalb - Best Practices for Object-Oriented Programming
Arno Schoedl - From Iterators To Ranges — The Upcoming Evolution Of the Standard Library
Juan Pedro Bolivar Puente - Squaring the circle: value-oriented design in an object-oriented system
Kate Gregory - Naming is Hard: Let's Do Better
Kevlin Henney - Lambda? You Keep Using that Letter
Luna Kirkby - Mind the Bear Traps!
Mateusz Pusz - Rethinking the Way We Do Templates in C++ even more
Matt Godbolt - Correct by Construction: APIs That Are Easy to Use and Hard to Misuse
A.J. Orians - Improving Readability With Class Template Argument Deduction
Rainer Grimm - Concepts in C++20: A Evolution or a Revolution?
Viet Le - Packaging and distributing C++ for fun and profit
Vittorio Romeo - C++11/14 at scale - what have we learned?
Sandor Dargo - Undefined behaviour in the STL
Yuri Minaev - Paranoid's take on C++ code review
Andrzej Warzyński - How compilers work: introduction to LLVM passes
Anastasiia Kazakova - C++ ecosystem: the renaissance edition
Sy Brand - Live Compiler Development with Cross-Platform Tooling
Timur Doumler - Real-time STL
Tina Ulbrich and Niel Waldren - Pythonic C++
Boguslaw Cyganek - How accurate we are? A refresher on the floating-point computations and the standard libra
Clare Macrae - Quickly Testing Legacy C++ Code with Approval Tests
Danila Kutenin - C++ STL best and worst performance features and how to learn from th
```

Email solutions to pca@caphyon.com

See you in 2 weeks…

Don’t forget about your assignments

Homework

Email solutions to pca@caphyon.com