Me: Viktor Sehr
From: Stockholm, Sweden

viktor.sehr@gmail.com

http://www.kebnekaisethegame.com/

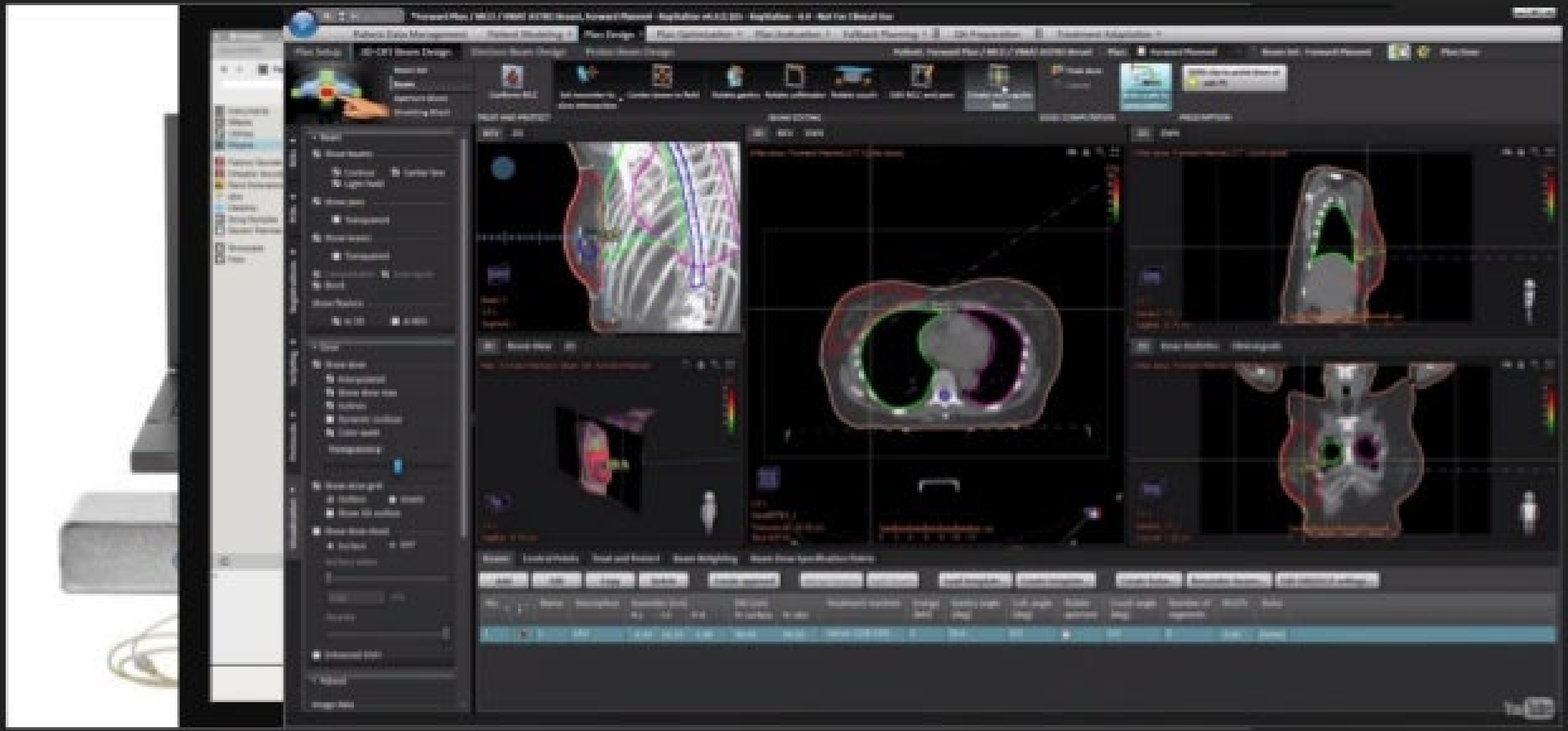
(Name of game will probably change later)

Professional experience

Mentice - Medical simulators for endovascular surgery (bugs = restart)

Propellerheads - Music making software (bugs = glitches in audio)

RaySearch - Radiation Therapy Planning Software (bugs = death)



My current job 1/2

Making my own skiing game for Android/iOS

100% C++

Working title: “Kebnekaise” (Sweden's highest mountain at 2099m)

Please buy it when it’s released :)

My current job 2/2

- I’m also co-writing a book on C++ for Packt Publishing
- Please buy that one too once released
- … and also one for your mother :)

C++ Advantages

- You can write everything from GUI down to high performance engines in the same code base
- Interacting between Java/C#/Whatever and C++ is boring
- Platform independent

Robustness
- Value semantics, C#/Java/etc is just a bunch of shared_ptr
- Const values, const member functions
- Ability to create libraries which are hard to misuse

Better than ever:
- Memory management is a thing of the past!
- Verbose syntax is also a thing of the past!

C++ Disadvantages

At a glance

- Compile times :(:(:(
- Manual labour to get around compile times (Why doesn’t IDE’s handle forward declarations and #includes?)
- Code completion is still quite bad (at least in Visual Studio 2017)
- Very few built-in libraries

Library handling

- Sometimes horrendously complicated if they require building
- Header only libraries are nice, but increases the compile time
- Modules are on the way :)

My #1 surprise when I started to work

Even professional applications consists of:

Spaghetti code (Although built on listener and weird inheritance instead of gotos)

Copy-Pasting

Unbelievably complex procedures for adding for example a resource

Initialization code is copied straight from some “hello world” tutorial on the internet

Timewasting debugging procedures

Flooded with bulk code which doesn’t add anything

Programmers accept working in a SLOW debug build

(... my own skiing game does quite a lot of this as well)

How do we make the codebase manageable?

Don’t waste time recompile/restart all the time

Don’t waste time develop in pure debug mode

Detect bugs as soon as possible

As much information as possible once a bug is detected

Debuggability - Project targets

(At least) three project targets (usually project defaults to two)

Debug

- Execute at least once in awhile to get the STL and runtime assertments

Produce

- This is the version developers and testers use!
- Your asserts are enabled
- As optimized as can be without compromising compile time

Release

- Maximum performance
- Long compilation times due to link time optimizations enabled
- No assertments at all

The very important ASSERT macro

The very important ASSERT macro

Your Assert macro will be your main bug-scout

The information the assert macro provides will be your main bug information source

Your assert macro will also disturb you with bug's your colleagues are (hopefully) resolving right now

Thus - make sure you have a good assert macro

ASSERTS

Store the values evaluated in the ASSERT!

ASSERT - Append messages

Should be ridiculously easy to provide information...

… otherwise people will not…

… and you spend more time debugging.

```
auto&& haystack = std::string("");
auto&& needle = std::string("");
ASSERT(haystack.find(needle) != std::string::npos);

// Very bad
if(haystack.find(needle) != std::string::npos) {
  LOG() << haystack;
  LOG() << needle;
  ASSERT(haystack.find(needle) != std::string::npos);
}
// Better...
ASSERT_MSG(haystack.find(needle) != std::string::npos, haystack << " " << needle);

// Almost there...
ASSERT(haystack.find(needle) != std::string::npos) << haystack << " " << needle;

// Finally!
ASSERT(haystack.find(needle) != std::string::npos) << haystack << needle;
```

ASSERT - Keep the factory going

Make every assert only fire ONCE...

...otherwise your colleagues ASSERTS will slow down your momentum.

Make it toggleable…

… otherwise you will get working with performance intense parts

… also people may avoid heavier assertments

(Also provide a callstack with some tool in your log)

ASSERT

```cpp
namespace expression_fetcher {
  class fetcher {
  public:
    auto is_stopped() const { return _stopped; }
    auto get_string() const {
      auto str = std::string{};
      for (const auto& val : _values)
        str += val + " ";
      return str;
    }
    template <typename T> auto grab(const T& ivalue) {
      if (!_stopped) {
        std::stringstream sstr;
        sstr << ivalue << " ";
        _values.push_back(sstr.str());
      }
    }
    auto stop() { _stopped = true; }
    auto reverse() { std::reverse(_values.begin(), _values.end()); }
  private:
    std::vector <std::string> _values;
    bool _stopped = false;
  };
  template <typename T> auto& operator% (fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator==(fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator!=(fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator< (fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator> (fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator>=(fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator<=(fetcher& ifetcher, const T& ivalue) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator&&(fetcher& ifetcher, const T& ivalue) { ifetcher.stop(); return ifetcher; }
  template <typename T> auto& operator||(fetcher& ifetcher, const T& ivalue) { ifetcher.stop(); return ifetcher; }

  template <typename T> auto& operator% (const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator==(const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator!=(const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator< (const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator> (const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator<=(const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator>=(const T& ivalue, fetcher& ifetcher) { ifetcher.grab(ivalue); return ifetcher; }
  template <typename T> auto& operator&&(const T& ivalue, fetcher& ifetcher) { ifetcher.stop(); return ifetcher; }
  template <typename T> auto& operator||(const T& ivalue, fetcher& ifetcher) { ifetcher.stop(); return ifetcher; }
}
```

```cpp
#define FETCH_EXPRESSION_VALUES(exp) \
[&](){ \
  using namespace expression_fetcher; \
  fetcher left, right; \
  left % exp; \
  exp % right; \
  right.reverse(); \
  return left.is_stopped() ? \
    left.get_string() + " " + right.get_string() :\
    left.get_string(); \
}().c_str()
```

ASSERT

```cpp
auto test_nice_assert() {
    int a = 12;
    int b = 13;
    int c = 14;
    int d = 15;
    NICE_ASSERT(a == b && c > d);
    NICE_ASSERT(a == b);
    NICE_ASSERT(a > b);
    return true;
}
```

C:\Users\viktor.sehr\Dropbox\Belarus_Project\PortisWindows\..\Stage\x64Debug\PortisWindows.exe

```
ASSERTION FAIL!
 Expression: a == b && c > d
 Values: 12  13   14  15
 Msg:
 File: test_nice_assert
 Func: c:\users\viktor.sehr\dropbox\belarus_project\sharedsource\corehard\corehard.cpp
 Line: 239

ASSERTION FAIL!
 Expression: a == b
 Values: 12  13
 Msg:
 File: test_nice_assert
 Func: c:\users\viktor.sehr\dropbox\belarus_project\sharedsource\corehard\corehard.cpp
 Line: 240

ASSERTION FAIL!
 Expression: a > b
 Values: 12  13
 Msg:
 File: test_nice_assert
 Func: c:\users\viktor.sehr\dropbox\belarus_project\sharedsource\corehard\corehard.cpp
 Line: 241
```

Contract programming

Provide a debug_validate() for every class

Contract programming

Refered to check invariant in the Java world

Validate your class is valid

The class should always be valid when entering and exiting a member function.

May NOT call any member function (well, you will get a stack overflow if so you will notice)

Accessed via a macro so no cost in release

Contract programming

```
struct Boat {
  Boat(std::string iname) : name_(iname) {
    DEBUG_VALIDATE();
  }
  ~Boat() { DEBUG_VALIDATE(); }
  auto debug_validate() const {
    ASSERT(speed_ >= 0.0f);
  }
  auto get_speed() const {
    DEBUG_VALIDATE(); // Only executes once
    return _speed;
  }
  auto set_speed(float ispeed) {
    DEBUG_VALIDATE(); // Executes at start and end of function
    _speed = ispeed;
  }

  float _speed = 0.0f;
  std::string _name;
};
```

Contract programming - Implement the macro

```cpp
template <typename T>
struct scope_exit{
  scope_exit(T&& ifunc) : func_(std::move(ifunc)) {}
  scope_exit(const scope_exit&) = delete;
  scope_exit(scope_exit&&) = delete;
  auto operator=(const scope_exit&)->scope_exit& = delete;
  auto operator=(scope_exit&&) -> scope_exit& = delete;
  ~scope_exit() { func_(); }
private:
  T func_;
};

template <typename T>
auto make_scope_exit(T&& ifunc) -> scope_exit<T>&& {
  return scope_exit<T>(std::move(ifunc));
}

#define DEBUG_VALIDATE() \
debug_validate(); \
auto&& validate_scope_exit = make_scope_exit([this]() { \
  if(!std::is_const_v<decltype(this)>) { \
    this->debug_validate(); \
  } \
});
```

Contract programming - Detect broken design

If you cannot call DEBUG_VALIDATE() in a member function...

… your design is a failure

… your coworkers will not understand your class

… redesign!

Contract programming - Detect broken design

```cpp
class Node {
public:
  using ItemType = int;

  Node(Node* iparent) : _parent(iparent) {}

  auto debug_validate() const {
    if (_children.first != nullptr) {
      ASSERT(_items.empty());
    }
    const auto num_children =
      (_children.first != nullptr ? 1 : 0) +
      (_children.second != nullptr ? 1 : 0);
    ASSERT(num_children == 0 || num_children == 2);
  }

  auto has_children() const -> bool { return _children.first != nullptr; }
  auto add_item(const ItemType& iitem) {
    DEBUG_VALIDATE();
    const auto th = 4;
    if (_items.size() < th) {
      _items.push_back(iitem);
      return;
    }
    if (_items.size() > th && !has_children()) {
      split_node();
    }
    for (auto&& item : _items) _children.first->_items.push_back(item);
    _items.clear();
  }
```

```cpp
  auto split_node() -> void {
    DEBUG_VALIDATE();
    _children.first = new Node(this);
    _children.second = new Node(this);
  }

private:
  std::vector<ItemType> _items;
  std::pair<Node*, Node*> _children = { nullptr, nullptr };
  Node* _parent = nullptr;
};
```

Reduce code size with reflection

Instantly get rid of thousands of lines of bulk code, half written classes and weird tuple inheritances.

And you can log your classes in your asserts :)

But C++ doesn't have reflection?

A tuple of references to your class members is enough for most cases

You can implement it easily with std::tie or use Boost hana/pfr

Reduce code size with reflection

```
class Fish_NoReflect {
public:
  auto operator==(const Fish_NoReflect& iother) const -> bool {
    return
      _name == iother._name &&
      _weight == iother._weight;
  }
  auto operator!=(const Fish_NoReflect& iother) const -> bool {
    return !(*this == iother);
  }
  auto operator<(const Fish_NoReflect& other) const -> bool {
    if (_name != other._name) return _name < other._name;
    return _weight < other._weight;
  }
  std::string _name;
  float _weight;
};

// Serialization
template <typename T>
auto& operator<<(std::ostream& ostr, const Fish_NoReflect& ifish) {
  ostr << ifish._name;
  ostr << " ";
  ostr << ifish._weight;
  return ostr;
}
```

```
class Fish {
public:
  auto reflect() const { return std::tie(_name, _weight); }
  std::string _name;
  float _weight;
};
```

Reduce code size with reflection

```
template <typename T> using has_reflect = decltype(&T::reflect);
template <typename T> constexpr bool is_reflectable_v = std::experimental::is_detected<has_reflect, T>::value;

template <typename T, bool HasReflect = is_reflectable_v<T>>
auto operator==(const T& a, const T& b) -> std::enable_if_t<HasReflect, bool> {
  return a.reflect() == b.reflect();
}

template <typename T, bool HasReflect = is_reflectable_v<T>>
auto operator!=(const T& a, const T& b) -> std::enable_if_t<HasReflect, bool> {
  return a.reflect() != b.reflect();
}

template <typename T, bool HasReflect = is_reflectable_v<T>>
auto operator<(const T& a, const T& b) -> std::enable_if_t<HasReflect, bool> {
  return a.reflect() < b.reflect();
}

template <typename T, bool HasReflect = is_reflectable_v<T>>
auto operator<<(std::ostream& ostr, const T& ivalue) -> std::enable_if_t<HasReflect, std::ostream&> {
  tuple_for_each(ivalue.reflect(), [&ostr](const auto& ival) {
    ostr << ival << " ";
  });
  return ostr;
}
```

Reduce code size with reflection

```cpp
auto test_reflection() {
    auto fish_a = Fish{ "Shark", 32.0f };
    auto fish_b = Fish{ "Shark", 32.0f };
    auto fish_c = Fish{ "Salmon", 42.0f };
    std::cout << fish_a << std::endl;
    std::cout << fish_b << std::endl;
    std::cout << fish_c << std::endl;
    std::cout << (fish_a == fish_b) << std::endl;
    std::cout << (fish_a == fish_c) << std::endl;
    return true;
}
```

C:\Users\viktor.sehr\Dropbox\Be...

```
Shark 32
Shark 32
Salmon 42
1
0
```

Reduce code size with reflection

And a final note, **don’t compromise.**

A class is either reflectable or not, don’t reflect every member but one because that’s all you want in your operator==.

```
class Fish_BadReflection {
public:
    auto reflect() const { return std::tie(_weight); }
    std::string _name;
    float _weight;
};
```

Step backwards in debugger

You cannot step backwards...

… but you can write your code to make it almost possible

Declare const new variables instead of mutating variables

Of course, for optimization reasons, larger objects have to be mutated

Step backwards in debugger

```
template <typename ValueType>
bool
HitLineTriangle(const math::Line3<ValueType>& line, const math::Triangle3<ValueType>& triangle) {
  // Data
  ASSERT(triangle.defined());
  const auto& v0 = triangle.v0();
  const auto& v1 = triangle.v1();
  const auto& v2 = triangle.v2();
  const auto& p0 = line.start_;
  const auto& p1 = line.stop_;
  // Calculate
  const auto& p1_sub_p0 = p1 - p0;
  const auto n = (v1 - v0).cross(v2 - v0);
  const auto ri_denominator = dot(n, p1_sub_p0);
  if (ri_denominator == 0) // Parallel
    return false;
  const auto ri = math::dot(n, (v0 - p0)) / ri_denominator;
  const auto pi = p0 + p1_sub_p0*ri;
  const auto w = pi - v0;
  const auto u = v1 - v0;
  const auto v = v2 - v0;
  const auto dot_u_v = math::dot(u, v);
  const auto dot_u_u = math::dot(u, u);
  const auto dot_w_u = math::dot(w, u);
  const auto dot_w_v = math::dot(w, v);
  const auto dot_v_v = math::dot(v, v);
  const auto denominator = (dot_u_v*dot_u_v) - (dot_u_u * dot_v_v);
  ASSERT(0 != denominator);
  const auto one_div_denominator = 1 / denominator;
  const auto si = (dot_u_v*dot_w_v - dot_v_v*dot_w_u) * one_div_denominator;
  const auto ti = (dot_u_v*dot_w_u - dot_u_u*dot_w_v) * one_div_denominator;
  return si >= 0 && ti >= 0 && (si + ti <= 1);
}
```

Pure functions are always good design

Also “almost pure” are good design :)

Programming functional has become quite a hype in recent years

- Easy to test (regardless of how you use unit tests)
- If it’s too complicated to debug, you can always rewrite it from scratch!
- ...therefore keep your interface clean!
- When optimizing a pure function, it can be verified against the old version!

Pure functions - Validate new code

```
auto math::DistancePointRect(const math::Vec2f& ipoint, const math::Rectf& irect) -> float {
  // New optimized distance, but we are not 100% sure it works
  const auto& lines = util::make_array(
    math::Line2f{ math::Vec2f{ irect.minx(), irect.miny() }, math::Vec2f{ irect.maxx(), irect.miny() } },
    math::Line2f{ math::Vec2f{ irect.minx(), irect.maxy() }, math::Vec2f{ irect.maxx(), irect.maxy() } },
    math::Line2f{ math::Vec2f{ irect.minx(), irect.miny() }, math::Vec2f{ irect.minx(), irect.maxy() } },
    math::Line2f{ math::Vec2f{ irect.maxx(), irect.miny() }, math::Vec2f{ irect.maxx(), irect.maxy() } }
  );

  const auto distances = lines | pip::transformed([ipoint](const auto& iline) {
    return DistancePointLineSegment(ipoint, iline);
  });
  const auto min_dist = *std::min_element(distances.begin(), distances.end());
  const auto dist = irect.surrounds(ipoint) ?
    -1.0f * std::abs(min_dist) :
    std::abs(min_dist);

  // Old distance calculation
  ASSERT(dist == [&]() {

    auto old_dist = irect + etc etc etc
    ...slow code which works...
    return old_dist;

  }());
  return dist;
}
```

Building a maintainable codebase

Distinguish between libraries and "Business logic"

Libraries:

- Contains the smartness and C++ technicalities

"Business logic"

- Every smartness is wrapped in your libraries
- No copy-constructor, copy-assignment, destructor etc
- Should not require deep C++ knowledge to work with
- Resembles more or less a scripting language

Building a maintainable codebase

How can we implement this class without adding custom copy-constructor etc?

```
class Antler {...};
class Moose {
public:
  auto set_antler(const Antler& iantler) -> void { ???; }
  auto remove_antler() -> void { ??? }
  auto has_antler() const -> bool { return ???; }
  auto get_antler() const -> const Antler& { ???; }
private:
  ??? _antler;
};
```

Building a maintainable codebase

```
// Before C++17



class Antler {...};
class Moose {
public:
  auto set_antler(const Antler& iantler) -> void { _antler.clear(); _antler.push_back(iantler); }
  auto remove_antler() { _antler.clear(); };
  auto has_antler() const -> bool { return _antler.size() > 0; }
  auto get_antler() const -> const Antler& { return _antler.front(); }

private:
  std::vector<Antler> _antler;
};
```

Building a maintainable codebase

```
// C++17

class Antler {...};
class Moose {
public:
  auto set_antler(const Antler& iantler) -> void { _antler = iantler; }
  auto remove_antler() { _antler = {}; };
  auto has_antler() const -> bool { return _antler.is_initialized(); }
  auto get_antler() const -> const Antler& { return *_antler; }

private:
  std::optional<Antler> _antler;
};
```

Use wrappers!

- **Wrappers are good!**
- Communicate with types and function names
- Wrap even if you don’t have time for implementation, the interface is enough to start with!
- Can be used to get instant asserts
- Wrap STL algorithms

The wrappers - Examples

- ranged_value
- synchronized_value
- non_empty

The wrappers - non_empty

Empty containers are like null pointers
Wrap algorithms requiring containers with size > 0

```
template <typename C>
auto mean_value(const non_empty<C>& icontainer) {
  auto sum = std::accumulate(icontainer.begin(), icontainer.end(), 0);
  return sum / icontainer.size();
}

auto test_non_empty() {
  auto&& vec = std::vector<int>{ 1, 2, 3 };
  auto m = mean_value(make_non_empty(vec));
  return true;
}
```

```
template <typename C>
class non_empty {
public:
  non_empty(const C& ibase) : _base(ibase) {
    NICE_ASSERT(!_base.empty());
  }
  auto begin() const { return _base.begin(); }
  auto end() const { return _base.end(); }
  auto size() const { return _base.size(); }
private:
  const C& _base;
};

template <typename C>
auto make_non_empty(const C& icontainer) { return non_empty<C>{icontainer}; }
```

The wrappers - ranged_value

```cpp
template <typename T, int64_t MinValue, int64_t MaxValue>
class ranged_value {
public:
  ranged_value(){}
  ranged_value(const T& ivalue) : _value(ivalue) { DEBUG_VALIDATE(); }
  ~ranged_value() { DEBUG_VALIDATE(); }
  auto debug_validate() const {
    ASSERT(_value >= static_cast<T>(MinValue));
    ASSERT(_value <= static_cast<T>(MaxValue));
  }
  auto reflect() const { DEBUG_VALIDATE(); return std::tie(_value); }
  auto operator=(const T& ivalue) { DEBUG_VALIDATE(); _value = ivalue; }
  operator const T& () const { DEBUG_VALIDATE(); return value_; }
  auto operator+(const T& iother) const { DEBUG_VALIDATE();  return ranged_value(_value + iother); }
  // operator-
  // operator*
  // operator/
  // ...etc
private:
  T _value = static_cast<T>(MinValue);
};
```

The wrappers - good with lambdas

```cpp
template <typename ValueType, typename MutexType = std::mutex>
class synchronized_value {
public:
  using value_type = ValueType;
  using mutex_type = MutexType;
  synchronized_value() {}
  synchronized_value(ValueType&& ivalue)
  : value_(std::move(ivalue)) {}
  auto& operator=(ValueType ivalue) {
    auto&& guard = std::lock_guard<mutex_type>(mutex_);
    (void)(guard);
    value_ = std::move(ivalue);
    return *this;
  }
  template <typename Functor>
  auto mutate(Functor ifunctor) -> decltype(auto) {
    auto&& guard = std::lock_guard<mutex_type>(mutex_);
    (void)(guard);
    return ifunctor(value_);
  }
  template <typename Functor>
  auto observe(Functor ifunctor) -> decltype(auto) {
    auto&& guard = std::lock_guard<mutex_type>(mutex_);
    (void)(guard);
    return ifunctor(std::cref(value_));
  }

private:
  ValueType value_;
  MutexType mutex_;
};
```

```cpp
auto test_sync() {

  using vector_type = std::vector<int>;
  using sync_vector_type = synchronized_value<vector_type>;

  auto&& sync_vec = sync_vector_type{};
  sync_vec.mutate([](auto& ivec) {
    ivec.push_back(3);
  });

  auto size = sync_vec.observe([](const vector_type& ivec) {
    return ivec.size();
  });
  return true;
}
```

The wrappers - wrap STL algorithms

The begin/end philosophy is exhausting

Wrapping an algorithm into your little library takes a matter of seconds

Add minor abbreviations to the STL algorithms:

```
auto contains(Container, Value) -> bool;

auto find_or_fail(Container, Value) -> const ValueType&;

auto index_of(Container, Value) -> size_t;

etc
```

Questions?

Contact: viktor.sehr@gmail.com

Questions?

Contact: viktor.sehr@gmail.com

Questions?

Contact: viktor.sehr@gmail.com