STL Algorithms
Principles and Practice

Victor Ciura - Technical Lead, Advanced Installer
Gabriel Diaconița - Senior Software Developer, Advanced Installer

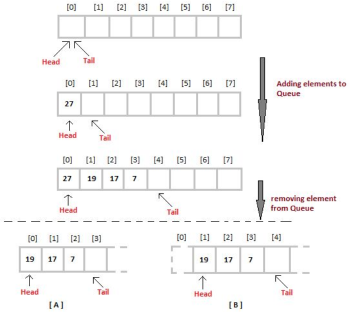
www.advancedinstaller.com

Winter 2017

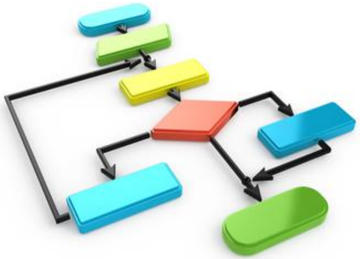
Agenda

Part 0: STL Background

Part 1: Containers and Iterators



Part 2-3: STL Algorithms Principles and Practice



Part 4: STL Function Objects and Utilities

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**

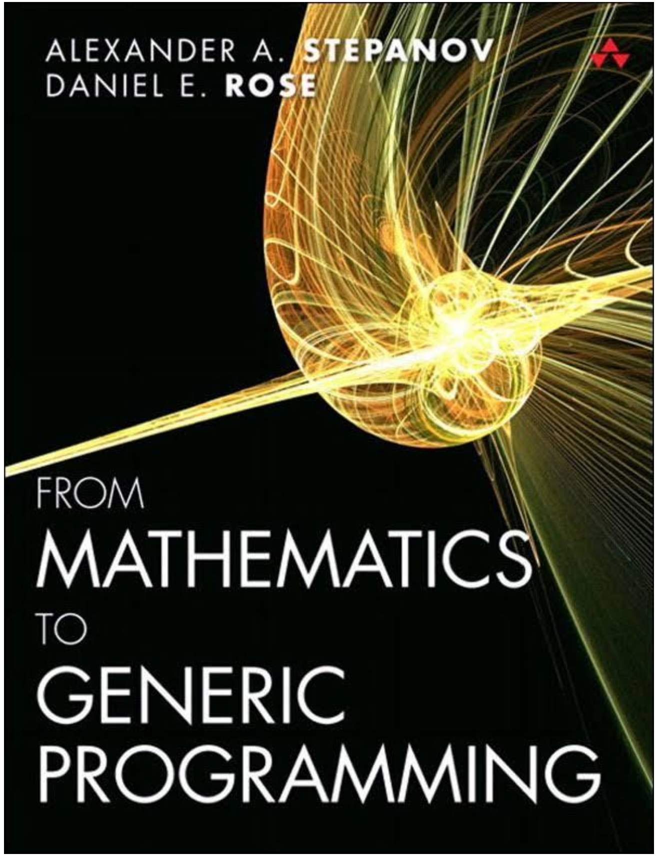
Alexander Stepanov (2002),

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)

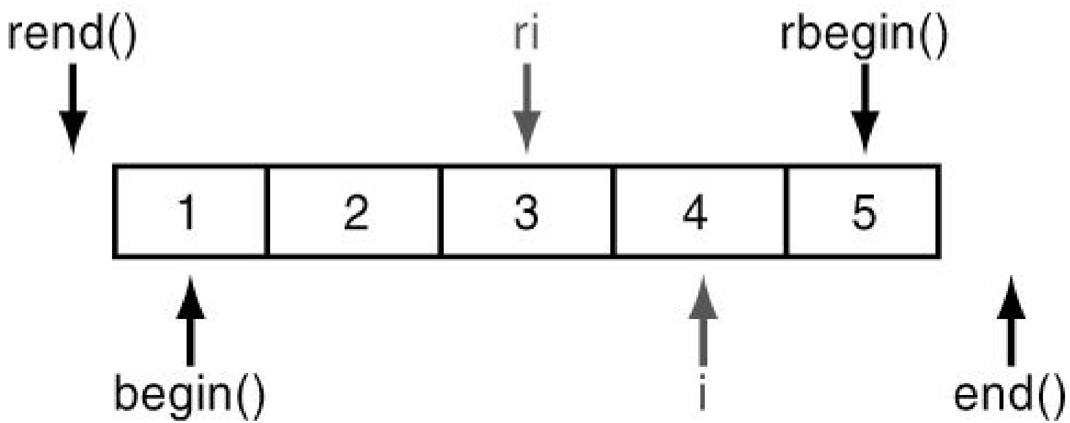
STL Iterators

- **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);`



SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << C approach >>

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

Output: 1 549 3 52 6

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

Can you spot any issues with this code?

Output: ???

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

Old code forgotten during refactoring. Compiler will catch this

SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << STL Iterator approach >>

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

Output: 1 549 3 52 6

```
vector<int> numbers = { 1, 549, 3, 52, 6 };
for (auto it = numbers.rbegin(), endIt = numbers.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

SAMPLE: C style iteration vs STL Iterators

Scenario: Refactor existing code so that is prints numbers in reverse order << C++11 range-for approach >>

```
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

Can you spo s with this

reverse() is an iterator adapter, which will be introduced shortly

Iterator Adaptors

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;

STL + Lambdas:

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

Range-for, using 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 at: **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; });
```

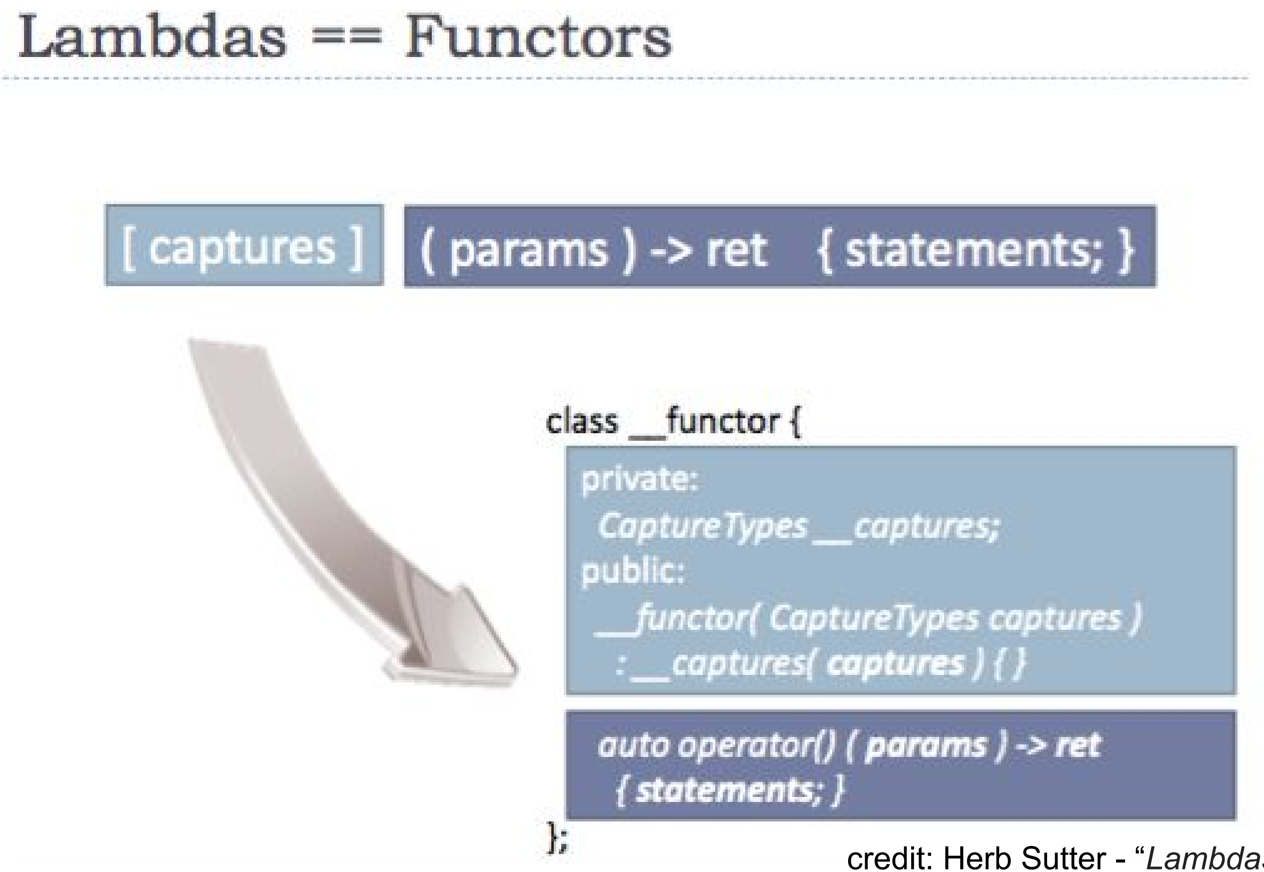
Lambda Functions

```
[ 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 :

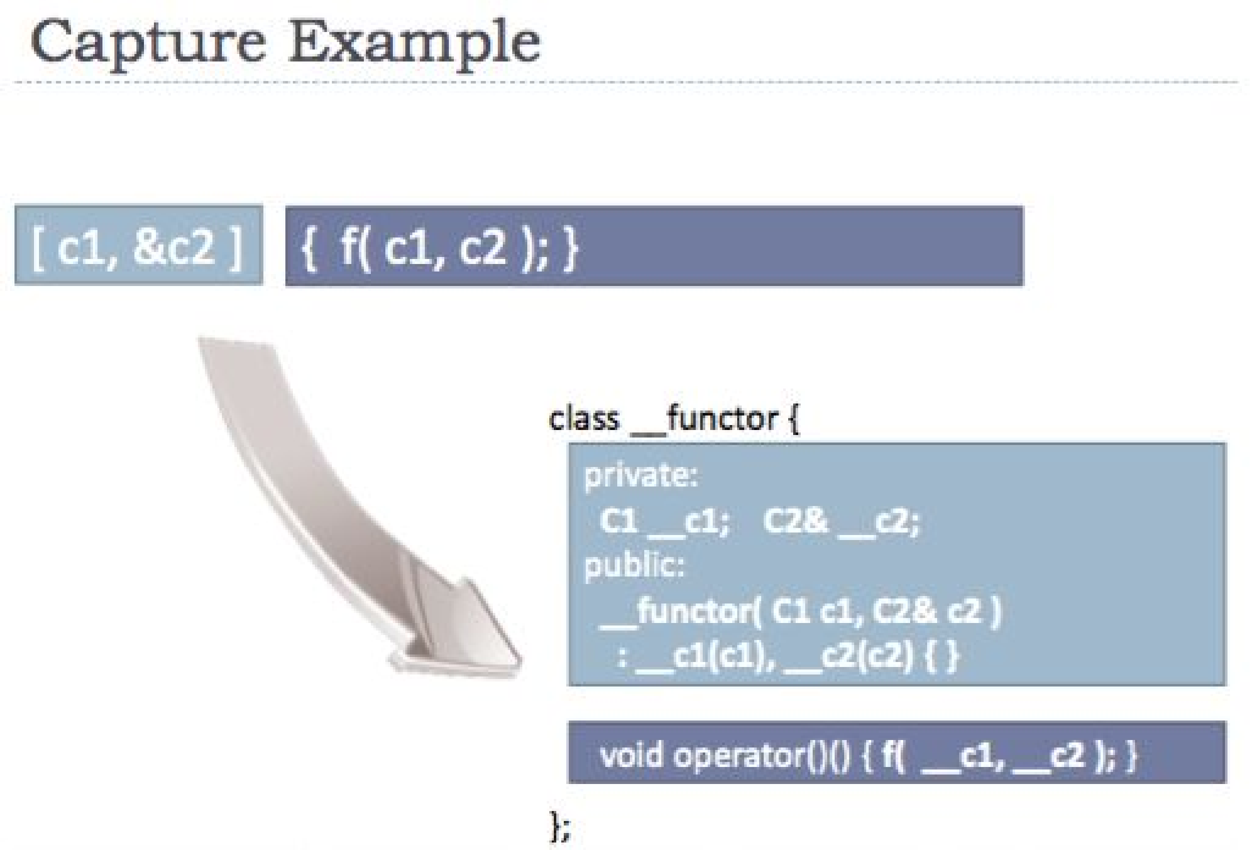
- **[a, &b]** where **a** is captured by ***value*** and **b** is captured by ***reference***.
- **[this]** captures the **this** pointer by ***value***
- **[&]** captures all automatic variables **used** in the body of the lambda by ***reference***
- **[=]** captures all automatic variables **used**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

Anatomy of A Lambda



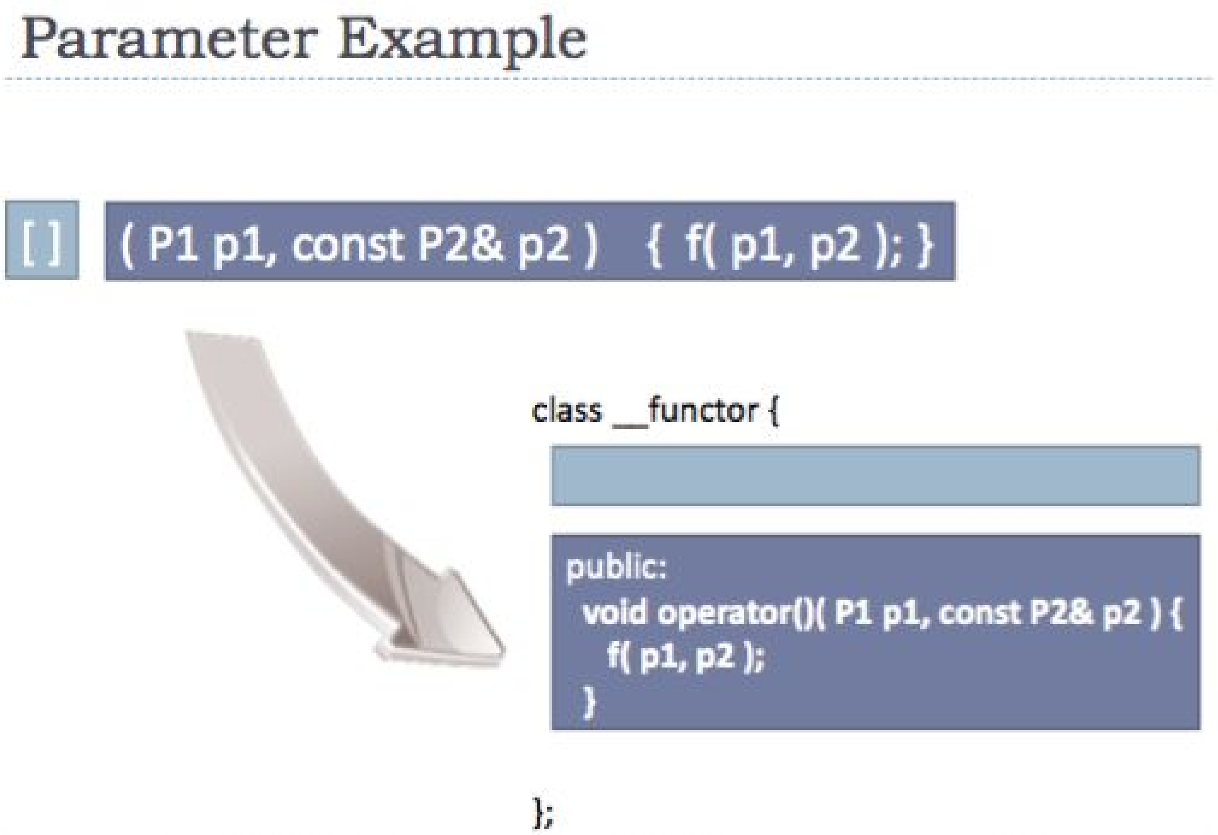
credit: Herb Sutter - “*Lambdas, Lambdas Everywhere*”
https://www.youtube.com/watch?v=rcgRY7sOA58

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:
namespace {
struct Lambda_247
{
  Lambda_247(unsigned int age) : minAge(age) {}
  bool operator()(const Person & p) { return p.age < minAge; }
  unsigned int minAge;
}; }

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

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 forced to think 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

Getting Inspired By Good Code

☞ ”To **write** better code, it's important to **read** good code.”

Jonathan Boccara

https://www.youtube.com/watch?v=kcfm7SKPn80

Here is how to find some great C++ code to get inspiration from:

www.fluentcpp.com/stl/ - a collection of resources on learning the **STL**

www.boost.org - the **Boost** libraries

theboostcpplibraries.com - the Boost book by Boris Schäling

www.reddit.com/r/cpp/ - the C++ sub-**reddit**

cppcast.com - the **podcast** by Rob Irving and Jason Turner

www.bfilipek.com/2017/01/cpp17features.html - good **blog** about **C++17** features

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 times).
- 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**

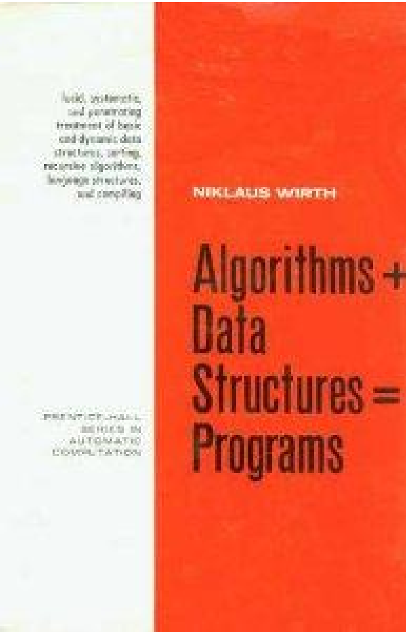
  (auto-vectorization, auto-parallelization, loop unrolling, dependency checking, etc.).

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.

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` http://en.cppreference.com/w/cpp/concept

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.

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`

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*

A type satisfies *Compare* if:

- it satisfies *BinaryPredicate* `bool comp(*iter1, *iter2);`
- it establishes a ***strict weak ordering*** relationship

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`

`{ partial ordering }`

Compare Examples

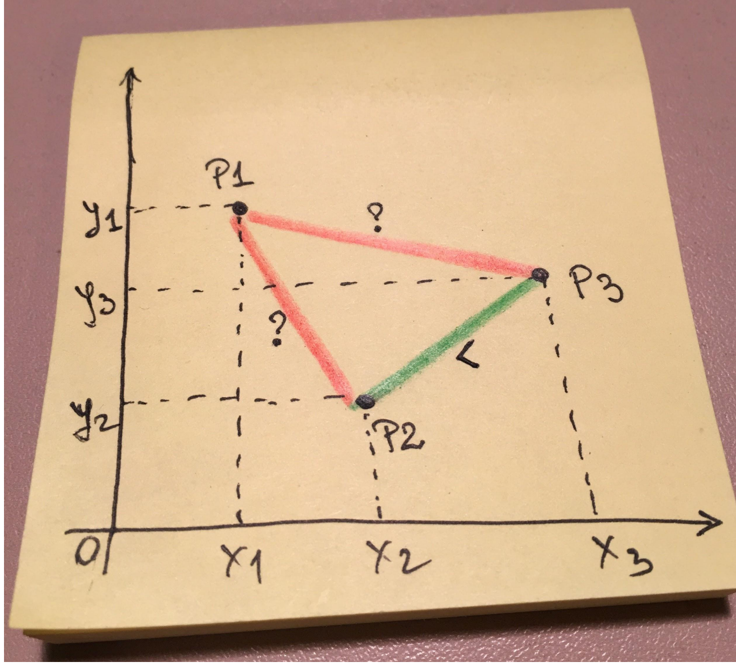
```
vector<string> v = { ... };

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: *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	`if a is equivalent to b and b is equivalent to c => a is equivalent to c`

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

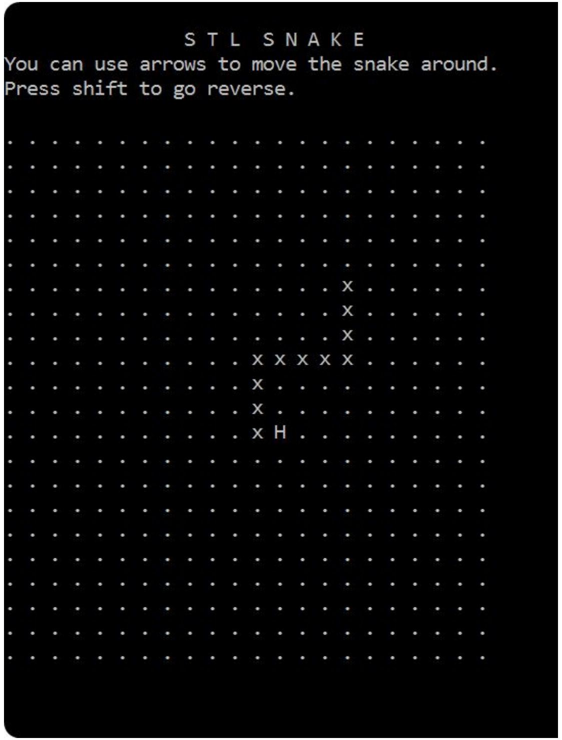
std::tuple< ... Types> - generalization of pair

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
- ...

Homework

We have a little game for you to refactor, using **STL**



Open with Visual Studio 2015/2017

Search for **#STL** blocks

Refactor C-style **#STL** blocks using valid STL code

Is the snake still snakin' & dyin' right?

Email solutions at: **gabriel.diaconita@caphyon.com**