STL Algorithms - Principles and Practice

“Prefer algorithm calls to hand-written loops.” - *Scott Meyers, "Effective STL"*

Victor Ciura - Technical Lead, Advanced Installer
Gabriel Diaconița - Senior Software Developer, Advanced Installer
http://www.advancedinstaller.com
CAPHYON

Why prefer to reuse (STL) algorithms?

Correctness

Fewer opportunities to write bugs (less code => less 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 reuse (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.

Why prefer to reuse (STL) algorithms?

Modern C++ (C++11/14 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 reuse (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**

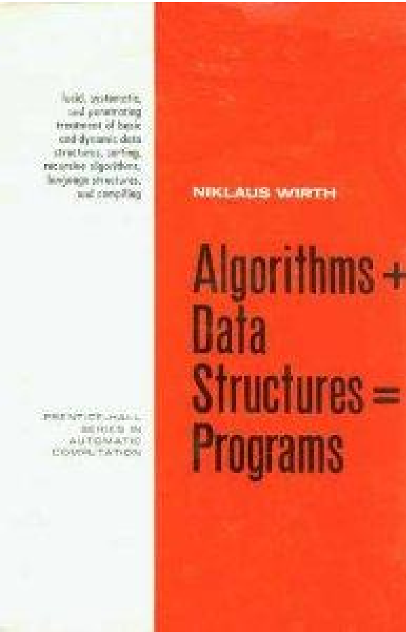
Why do we care ?

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

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



Optimization

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.

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
- specify correct and **complete interfaces** (eg. binary search should return the insertion point)
- look for interface **symmetry** (eg. stable_sort, stable_partition)
- **Iterators** are good (addresses are real in the machine) => ability to refer data through some handle
- Iterators should have fast comparison and dereferencing
- the STL library should be (easily) **extended** with other algorithms & data structures

Alexander Stepanov (2002), https://www.youtube.com/watch?v=COuHLky7E2Q

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)

Generic Programming Drawbacks

- abstraction penalty
- implementation in the interface
- early binding
- horrible error messages (*no formal specification* of interfaces, until C++17 **Concepts**)
- 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**

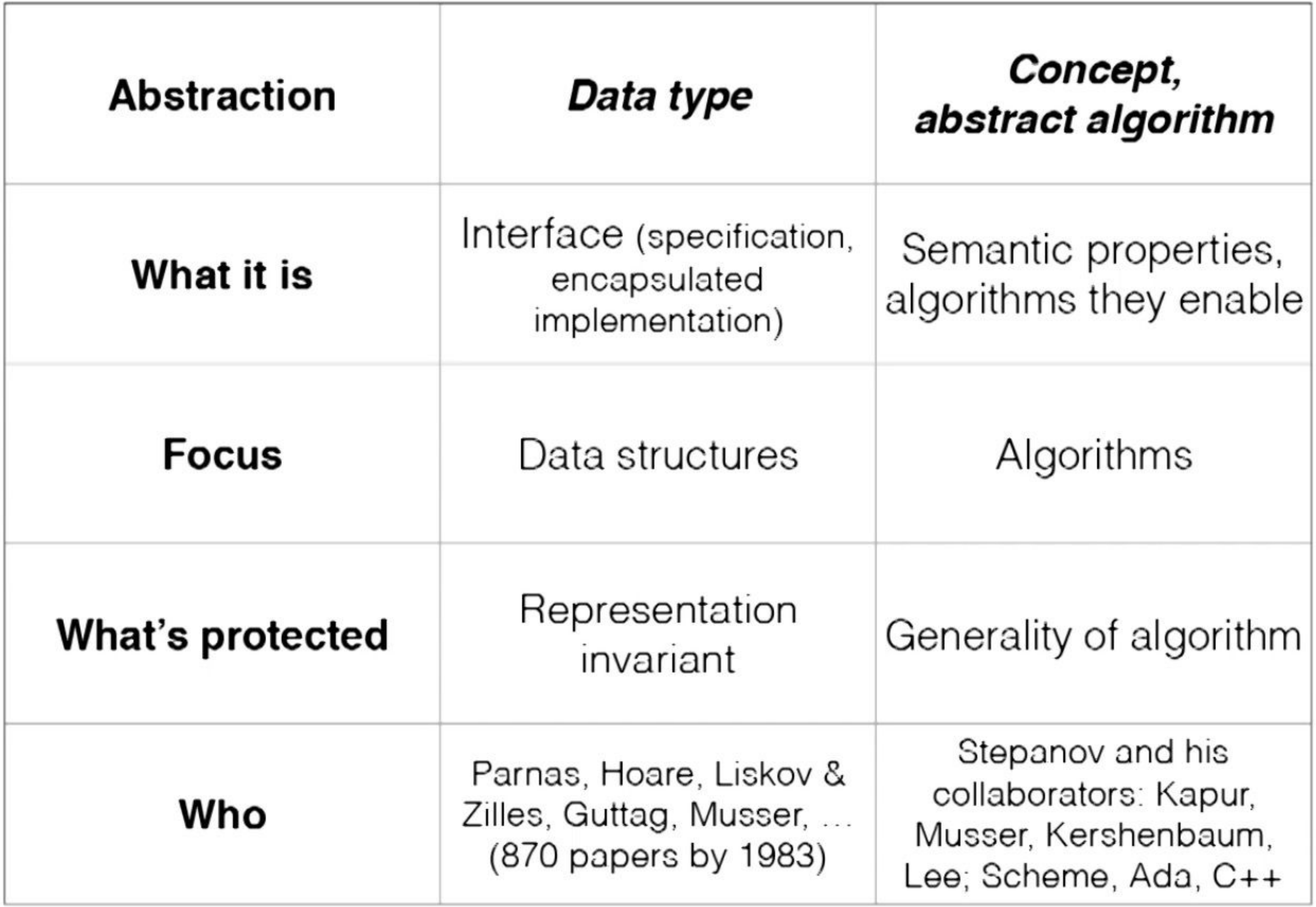
What Is A *Concept*, Anyway ?

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**

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



Abstraction	***Data type***	***Concept, abstract algorithm***
What it is	Interface (specification, encapsulated implementation)	Semantic properties, algorithms they enable
Focus	Data structures	Algorithms
What's protected	Representation invariant	Generality of algorithm
Who	Parnas, Hoare, Liskov & Zilles, Guttag, Musser, … (870 papers by 1983)	Stepanov and his collaborators: Kapur, Musser, Kershenbaum, Lee; Scheme, Ada, C++

Paul McJones

Template Constraints Using C++17 Concepts

An example: Balanced reduction

```
template<ForwardIterator I, BinaryOperation Op>
  requires EqualityComparable<ValueType<I>, Domain<Op> >()
Domain<Op> reduce(I it, DistanceType<I> n, Op  op)
// precondition: n != 0, "op" is  associative
{
  if (n == 1)
    return *it;

  DistanceType<I> h = n / 2;

  return op( reduce(it, h, op),
             reduce(it + h, n - h, op) );
}
```

*** For a better/efficient implementation of a generic **reduce**, see the longer (complex) implementation from ***Elements of Programming***, by Alexander Stepanov.

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

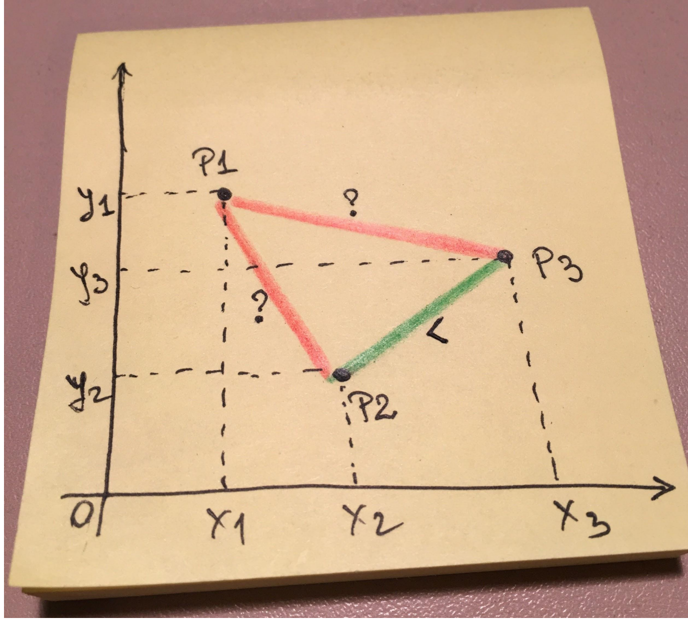
```
vector<string> v = { ... };

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
- ...

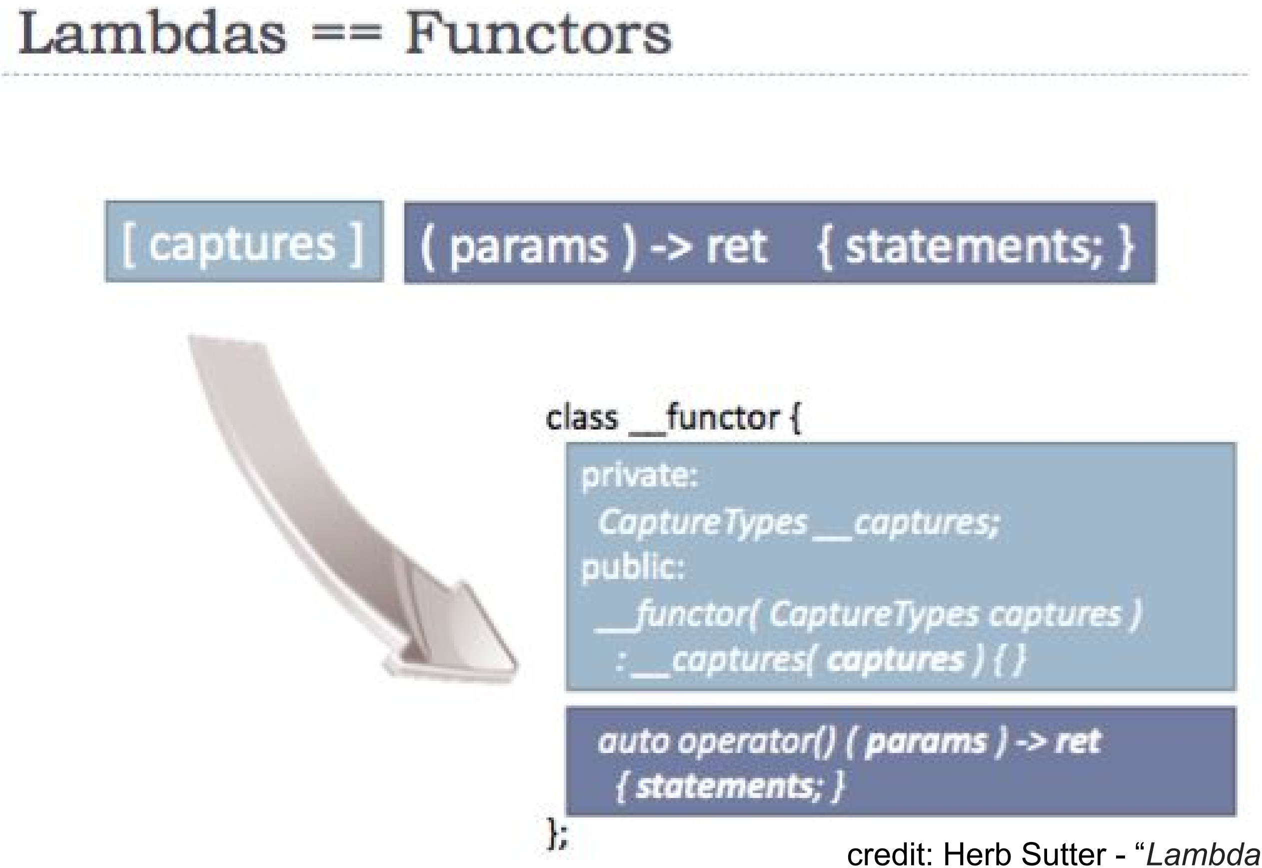
Prefer Function Objects or Lambdas to Free Functions

```
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.

Anatomy of A Lambda



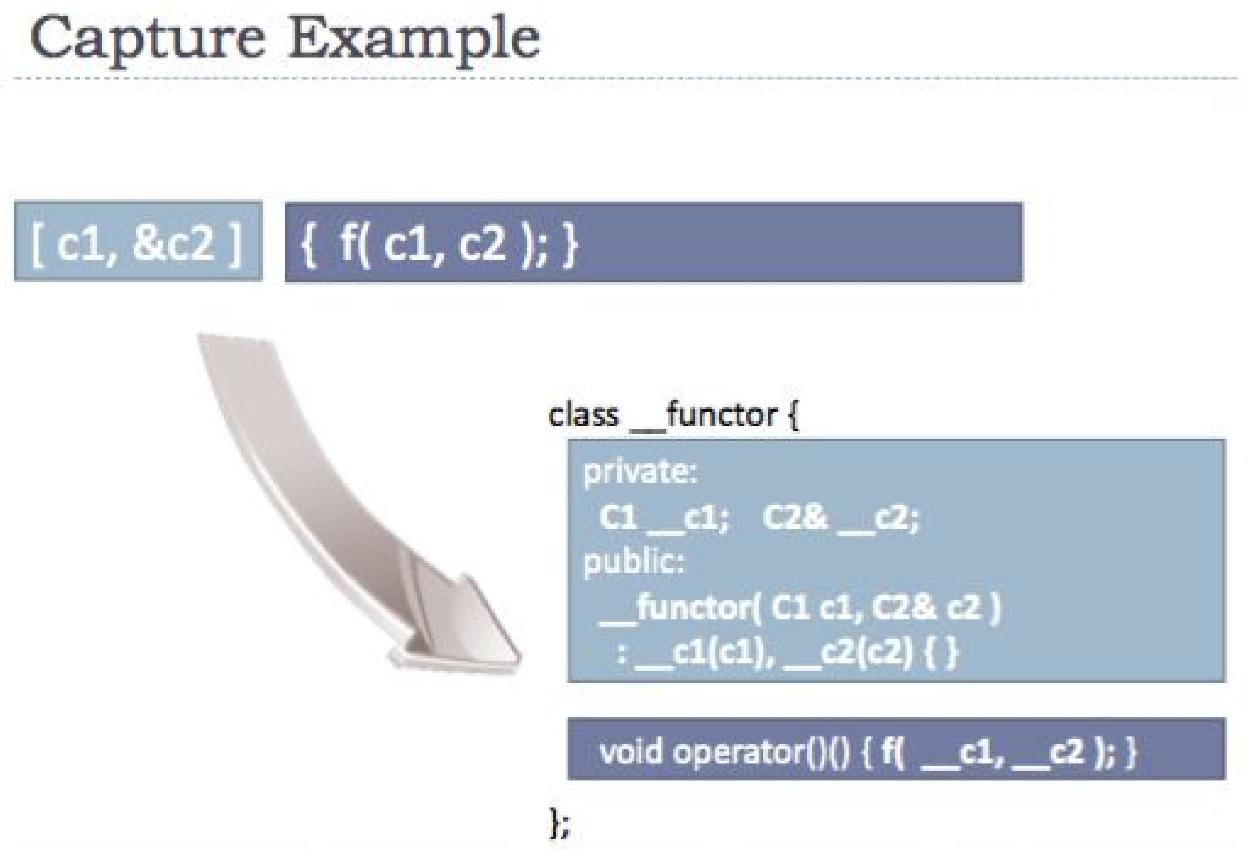
Lambdas == Functors

```
[ captures ]   ( params ) -> ret   { statements; }
```

```
class __functor {
  private:
    CaptureTypes __captures;
  public:
    __functor( CaptureTypes captures )
      : __captures( captures ) { }

    auto operator() ( params ) -> ret
      { statements; }
};
```

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

Anatomy of A Lambda



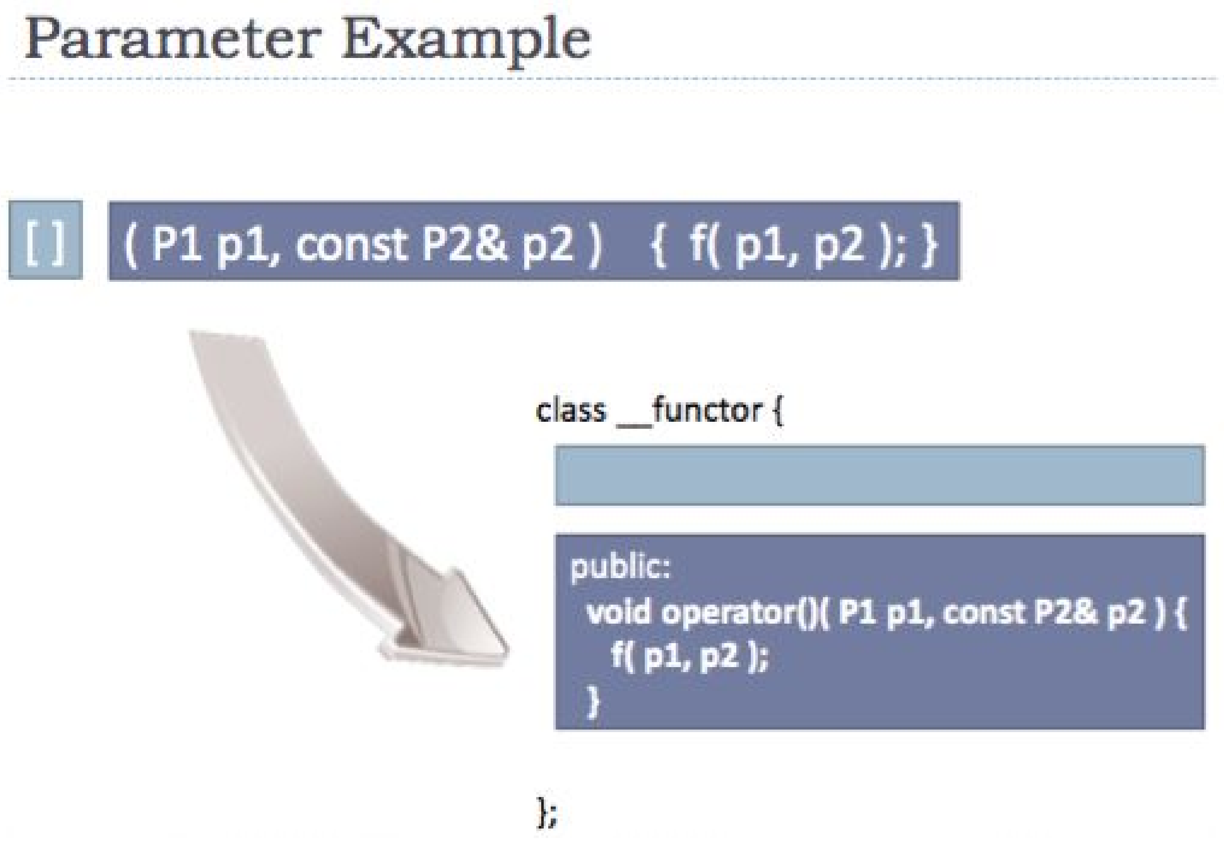
Capture Example

```
[ c1, &c2 ]  { f( c1, c2 ); }
```

```
class __functor {
  private:
    C1 __c1;   C2& __c2;
  public:
    __functor( C1 c1, C2& c2 )
      : __c1(c1), __c2(c2) { }

    void operator()() { f( __c1, __c2 ); }
};
```

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

Anatomy of A Lambda



Parameter Example

```
[ ]  ( P1 p1, const P2& p2 )   { f( p1, p2 ); }
```

```
class __functor {

public:
  void operator()( P1 p1, const P2& p2 ) {
    f( p1, p2 );
  }

};
```

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

Prefer Member Functions To Similarly Named Algorithms

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

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

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

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

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

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

Prefer Member Functions To Similarly Named Algorithms

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

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

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

Why ?

Prefer Member Functions To Similarly Named Algorithms

std::list**<>** specific algorithms

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

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

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

Binary search operations (on sorted ranges)

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

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

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

Binary search operations (on sorted ranges)

Counting elements equal to a given value

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

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

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

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

```
// iterative implementation
string bin(unsigned int n)
{
  string binStr;
  binStr.reserve(32);

  for (unsigned int i = 1u << 31; i > 0; i = i / 2)
    binStr += (n & i) ? "1" : "0";

  return binStr;
}
```

Is this implementation correct / complete ?

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

```
string binStr;
binStr.reserve(32);

// recursive implementation
void bin(unsigned int n)
{
  if (n > 1)
    bin( n/2 );

  binStr += (n % 2) ? "1" : "0";
}
```

Is this implementation correct / complete ?

Problem:

Print a non-zero unsigned 32-bit integer N into its binary representation, as a string.

* without leading zeros

```
// STL implementation
string bin(unsigned int  n)
{
  string binStr = std::bitset<32>(n).to_string();

  // erase leading zeros, if any
  binStr.erase(0, binStr.find_first_not_of('0'));
  return binStr;
}
```

Extend STL With Your Generic Algorithms

Eg.

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

Extend STL With Your Generic Algorithms

Eg.

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

Consider Adding Range-based Versions of STL Algorithms

```
namespace range {

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

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

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

}
```

Consider Adding Range-based Versions of STL Algorithms

```
vector<string> v = { … };

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

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

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

range::sort(v);

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

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

Eg.

```
std::map<int, string> m;

for_each( MakeSecondIterator(m.begin()), MakeSecondIterator(m.end()),
          [](const string & val) { cout << val << endl; } );

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

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
template <typename Iter>
class MapSecondIterator : public std::iterator<std::bidirectional_iterator_tag,
                                               typename Iter::value_type::second_type>
{
public:
  MapSecondIterator() {}
  MapSecondIterator(Iter aOther) : i(aOther) {}

  inline MapSecondIterator & operator++()    {...}
  inline MapSecondIterator   operator++(int) {...}
  inline MapSecondIterator & operator--()    {...}
  inline MapSecondIterator   operator--(int) {...}

  inline bool operator==(MapSecondIterator aOther) const {...}
  inline bool operator!=(MapSecondIterator aOther) const {...}

  inline reference operator*()   { return  i->second; }
  inline pointer   operator->()  { return &i->second; }

private:
  Iter i;
};
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

```
/**
 * Helper function that constructs
 * the appropriate iterator type based on ADL.
 */
template <typename Iter>
inline MapSecondIterator<Iter> MakeSecondIterator(Iter aIter)
{
  return MapSecondIterator<Iter>(aIter);
}

Eg.

  std::map<int, string> m;

  for_each( MakeSecondIterator(m.begin()), MakeSecondIterator(m.end()),
           [](const string & val) { cout << val << endl; } );
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

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

namespace std {

  template <typename T>
  auto begin(detail::IterateSecondWrapper<T> aWrapper)
  {
    return MakeSecondIterator( begin(aWrapper.mContainer) );
  }

  template <typename T>
  auto end(detail::IterateSecondWrapper<T> aWrapper)
  {
    return MakeSecondIterator( end(aWrapper.mContainer) );
  }
}
```

Iterator Adaptors

An iterator adapter, that helps iterate through a container's value_type pair SECOND value

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


Eg.

  std::map<int, string> m;

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

Iterator Adaptors

An iterator adapter that helps iterate a collection in reverse order

Eg.

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

C style:

```
  for (size_t 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

An iterator adapter that helps 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

An iterator adapter that helps 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);
  }
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

[C++98] ==> 2003-2012

```
// iterating table from within
MsiRowMap<Row>::RowMap::iterator  it = (*mRows)->begin();
MsiRowMap<Row>::RowMap::iterator end = (*mRows)->end();
for (; it != end; it++)
{
  DynamicFolderRow * row = (*it).second;
}

// iterating table from outside
auto_ptr<IMsiTable::RowIterator> it (mDynFolderTable.Begin());
auto_ptr<IMsiTable::RowIterator> end(mDynFolderTable.End());
for (; *it != *end; (*it)++)
{
  DynamicFolderRow * dynFolderRow =  static_cast<DynamicFolderRow *>(**it);
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

[C++11] ==> 2012-2016

```
// iterating table from within
auto it  = (*mRows)->begin();
auto end = (*mRows)->end();
for (; it != end; it++)
{
  DynamicFolderRow * row = (*it).second;
}

// iterating table from outside
auto iter(mDynFolderTable.Begin());
auto end (mDynFolderTable.End());
for (; *iter != *end; (*iter)++)
{
  DynamicFolderRow * dynFolderRow =  static_cast<DynamicFolderRow *>(**iter);
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

[C++14] ==> 2016-present

Using range-for loops.

```
// iterating table from within
for (DynamicFolderRow * row :  mRows)
{
   ...
}

// iterating table from outside
for (auto & row : mDynFolderTable)
{
  DynamicFolderRow * dynFolderRow =  static_cast<DynamicFolderRow *>(*row);
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

Using range-for loops.

```
namespace std
{
  template<typename Row>
  inline auto begin(MsiRowMap<Row> & aRows) // also overloaded for const
  {
    return MakeSecondIterator( aRows->begin() );
  }

  template<typename Row>
  inline auto end(MsiRowMap<Row> & aRows)   // also overloaded for const
  {
    return MakeSecondIterator( aRows->end() );
  }
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

Using range-for loops.

```
namespace std
{
  template<typename Row>
  inline auto begin(LazyAutoPtr<MsiRowMap<Row>> & aRows) // also overloaded for const
  {
    return MakeSecondIterator( (*aRows)->begin() );
  }

  template<typename Row>
  inline auto end(LazyAutoPtr<MsiRowMap<Row>> & aRows)   // also overloaded for const
  {
    return MakeSecondIterator( (*aRows)->end() );
  }
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

Using range-for loops.

```
namespace std
{
  inline auto begin(IMsiTable & aTable)
  {
    return aTable.Begin();
  }

  inline auto end(IMsiTable & aTable)
  {
    return aTable.End();
  }
}

class IMsiTable
{
  ...
  virtual ExternalRowIterator Begin() = 0;
  virtual ExternalRowIterator End() = 0;
}
```

Improving iteration through Advanced Installer MSI Tables

MSI Table iteration through the years: **2003 - 2016**

```
using RowIterator = IIterator<IMsiRow *>;
using RowIteratorUniquePtr = unique_ptr<RowIterator>;

/**
 * Iterator used for public access to table rows.
 */
struct ExternalRowIterator : public RowIteratorUniquePtr
{
 ExternalRowIterator(RowIteratorUniquePtr && aIter)  : RowIteratorUniquePtr(std::move(aIter))  {}
 ExternalRowIterator(ExternalRowIterator && aOther) : RowIteratorUniquePtr(std::move(aOther)) {}

 ExternalRowIterator(const ExternalRowIterator & aOther) = delete;
 ExternalRowIterator & operator=(const ExternalRowIterator & aOther) = delete;

 inline auto operator++()    { return ++(*get()); }
 inline auto operator++(int) { return (*get())++; }
 inline auto operator--()    { return --(*get()); }
 inline auto operator--(int) { return (*get())--; }

 inline friend bool operator!=(const ExternalRowIterator & aX, const ExternalRowIterator & aY){}
 inline friend bool operator==(const ExternalRowIterator & aX, const ExternalRowIterator & aY){}
};
```

Demo: Server Nodes

We have a huge network of server nodes.

Each server node contains a copy of a particular ***data*** `Value` (not necessarily unique).

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

{ *Assignable* + *Constructible* + *EqualityComparable* + *LessThanComparable* }

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

Eg.

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

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

Demo: Server Nodes

The network exposes the following APIs:

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

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

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

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

```
size_t GetNode(const Value & data) const;
```

STL Abuse

```
vector<int> vec = { ... };
int x = 3;
int y = 9;

vec.erase(
  remove_if(
    find_if(vec.rbegin(), vec.rend(),
            bind2nd(greater_equal<int>(), y)).base(),
    vec.end(),
    bind2nd(less<int>(), x)),
  vec.end());
```

Please don’t code like this !

Our own code. Calculating total number of unread messages.

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

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

Calculating total number of unread messages.

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

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

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

Our own code. Enabling move operation (up/down) for a List item in user interface

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

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

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

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

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

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

Our own code. Selecting attributes from XML nodes.

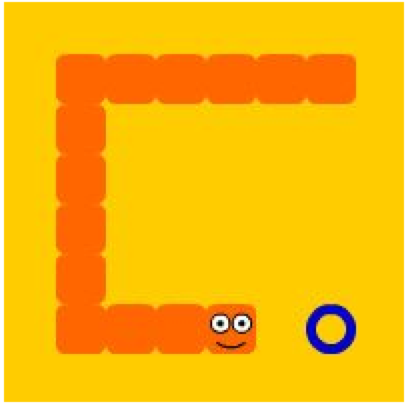
```
vector<XmlDomNode> childrenVector = parentNode.GetChildren(childrenVector);

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


// A good, range based for, alternative:

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


// Raw loop, see anything wrong?

for (unsigned int i = childrenVector.size(); i >= 0; i -= 1)
  childrenNames.insert(getNodeNameLambda(childrenVector[i]));
```

Time for coding fun!

On the USB stick you all received there's a Visual Studio project called **worm_stl** that's missing some key functionality.

Can you implement the required functionality using only STL algorithms?

Show us your solution!



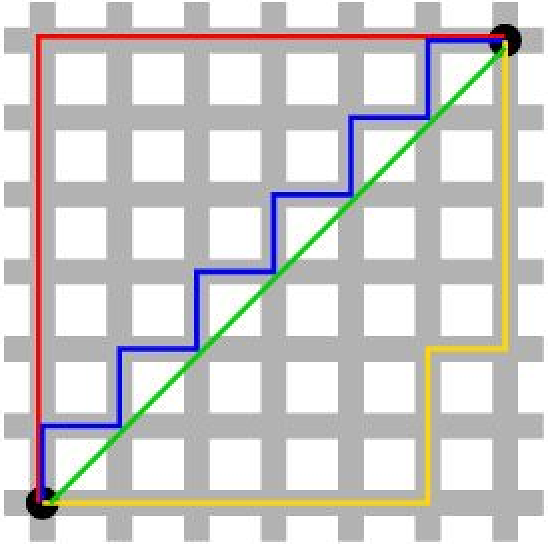
Manhattan Distance (Minkowski Geometry)

https://en.wikipedia.org/wiki/Taxicab_geometry

Taxicab geometry is often used in fire-spread simulation with square-cell, grid-based city maps like Manhattan (New York).
Fire hydrants are installed at each street block (square-cell). Each fire hydrant is connected to an underground water pipe.

We need to determine if we have enough redundancy for water supply pipes within each street block cluster.

Write a program that analyzes the water grid and reports all hydrants that are supplied by the same main water pipe within a maximum safety region of 4 blocks (Manhattan Distance).



Mathematical model of the problem:

"Given a 2D matrix with integer values, find duplicate values within a specified Manhattan distance of each other."

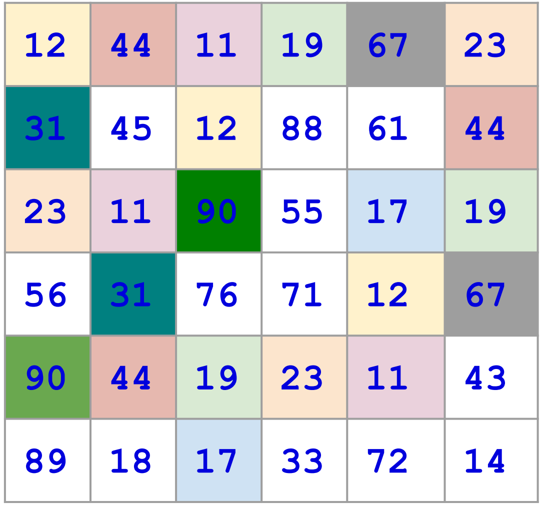
```
// Manhattan distance between 2 points
int ManhattanDist(Point p1, Point p2)
{
  return abs(p1.x - p2.x) + abs(p1.y - p2.y);
}
```

Manhattan Distance (example)

[Input]

Read an **M x N** matrix of integers from file or console.

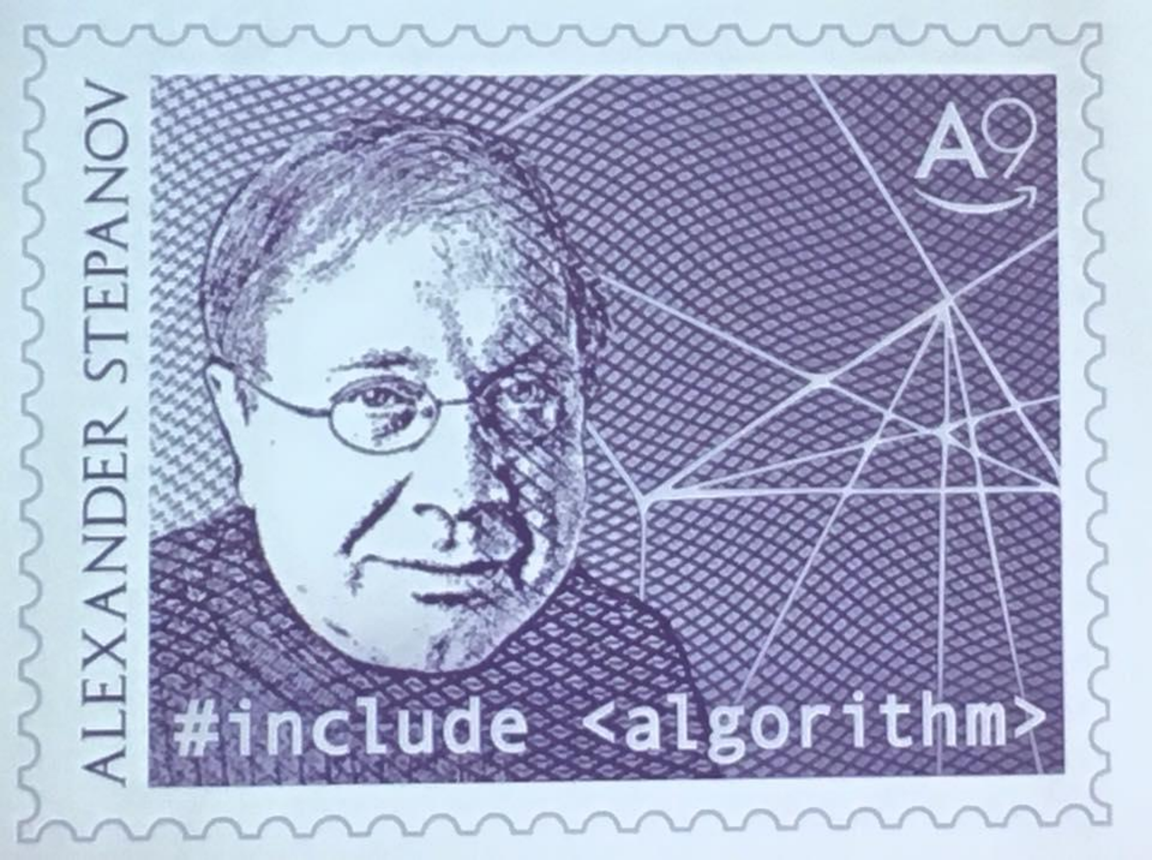
Read maximum Manhattan distance for duplicates: **4**



12	44	11	19	67	23
31	45	12	88	61	44
23	11	90	55	17	19
56	31	76	71	12	67
90	44	19	23	11	43
89	18	17	33	72	14

[Output]

```
Duplicates:
11 -> (0,2)(2,1)(4,4)  @ Manhattan distance <= 4 :  { (0,2)(2,1) }
12 -> (0,0)(1,2)(3,4)  @ Manhattan distance <= 4 :  { (0,0)(1,2) }, { (1,2)(3,4) }
17 -> (2,4)(5,2)
19 -> (0,3)(2,5)(4,2)  @ Manhattan distance <= 4 :  { (0,3)(2,5) }
23 -> (0,5)(2,0)(4,3)
31 -> (1,0)(3,1)       @ Manhattan distance <= 4 :  { (1,0)(3,1) }
44 -> (0,1)(1,5)(4,1)  @ Manhattan distance <= 4 :  { (0,1)(4,1) }
67 -> (0,4)(3,5)       @ Manhattan distance <= 4 :  { (0,4)(3,5) }
90 -> (2,2)(4,0)       @ Manhattan distance <= 4 :  { (2,2)(4,0) }
```



So sad to hear about Alex's retirement (Jan, 2016)...
I was looking forward to a few more years of excellent talks by a great scientist and educator.

Course Evaluation:
"STL Algorithms - Principles and Practice" by CAPHYON

Please take the survey:

https://www.surveymonkey.com/r/NTL23ZR

Survey results:

https://www.surveymonkey.com/results/SM-S5DPCCMW/

SurveyMonkey®