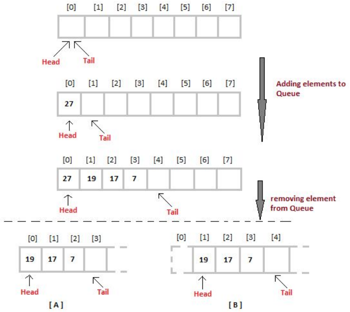
# STL - Principles and Practice

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

# Agenda

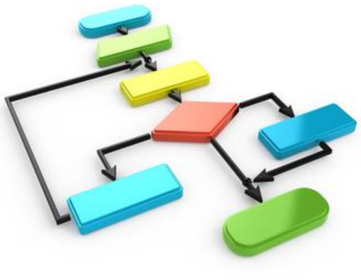
## Part 0: STL Intro.

## Part 1: Containers and Iterators




## Part 2: STL Function Objects and Utilities

## Part 3-4: STL Algorithms Principles and Practice



Part 3:

# STL Algorithms - Principles and Practice

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

# 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**

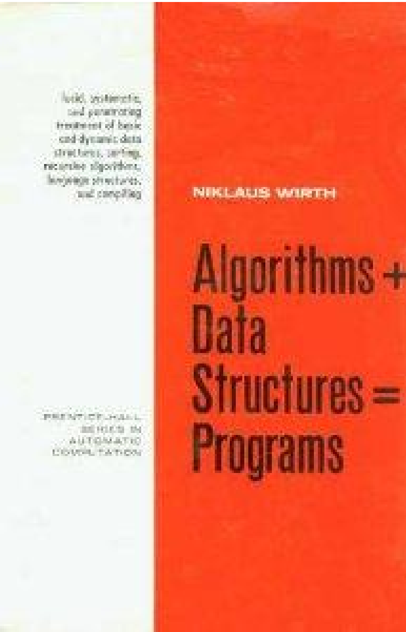
Why do we care ?

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.

Recap

# 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`

# 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;
});
```

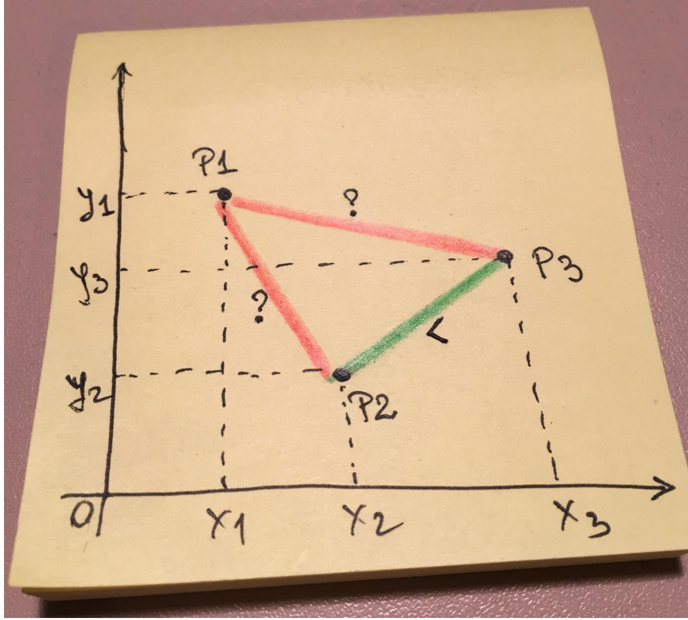
# *Compare* Examples

```
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 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);
```

# 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(); } );
```

# 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 !*

Extract algorithm **intermediate** results into **named** local variables (iterators, values, etc.)