



STL Algorithms - Principles and Practice

Victor Ciura - Technical Lead, Advanced Installer

Gabriel Diaconița - Senior Software Developer, Advanced Installer

<http://www.advancedinstaller.com>

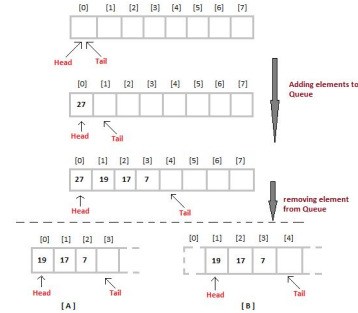
CAPHYON

Agenda

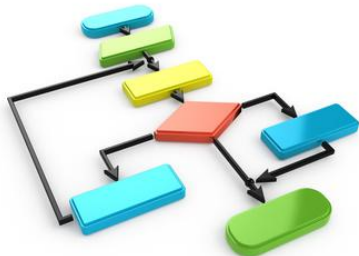
Part 0: STL Background



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**

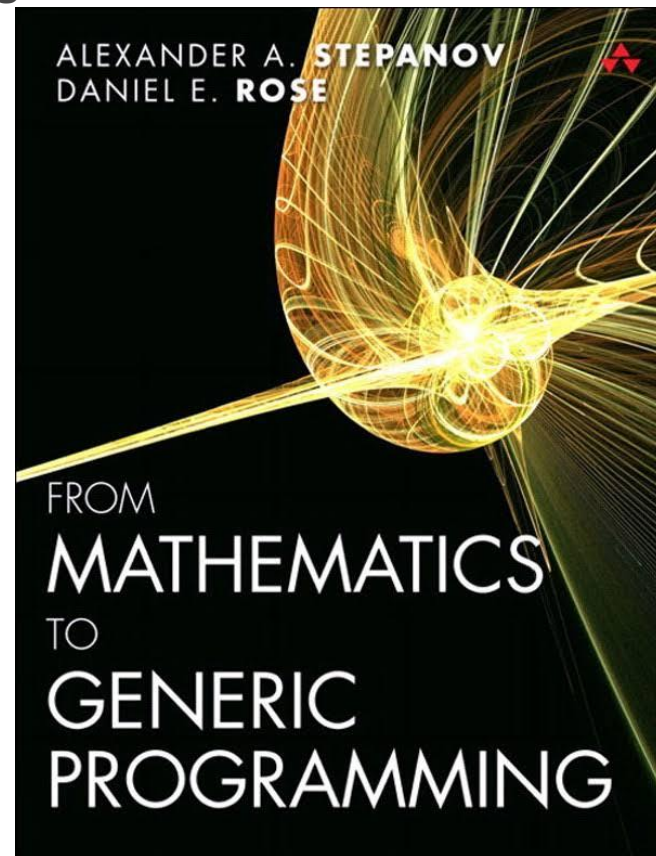
Alexander Stepanov (2002),

<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)

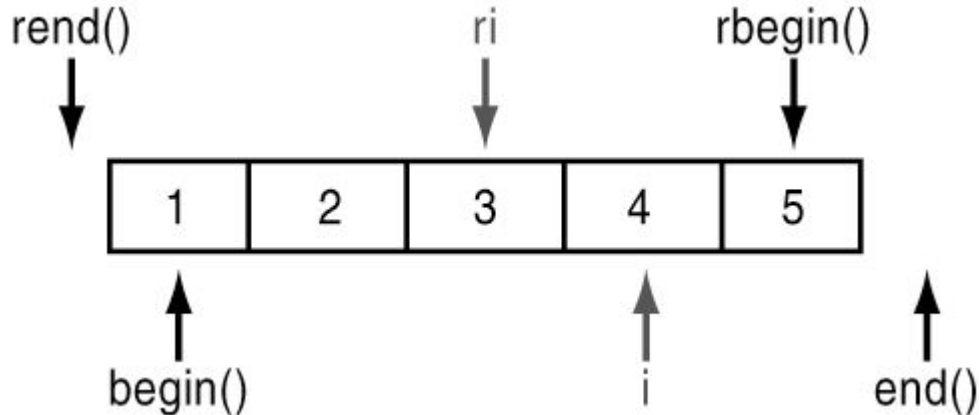
STL Iterators

- **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 it 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] << " ";
```

Output: ???

Can you spot any issues with
this code?

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 it 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 << *i << " ";
```

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 it 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 spot this with



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 adaptor:

```
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

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; }
```



Homework for the reader:

Using the same technique shown for `reverse()` iteration adaptor, implement `IterateFirst()` and `IterateSecond()` adaptors.

Function Objects Basics

```
template<class InputIt, class UnaryFunction>
void std::for_each( InputIt first, InputIt last, UnaryFunction func )
{
    for(; first != last; ++first)
        func( *first );
}
```

```
struct Printer // our custom functor for console output
{
    void operator()(const std::string & str)
    {
        std::cout << str << std::endl;
    }
};
```

```
std::vector<std::string> vec = { "STL", "function", "objects", "rule" };
```

```
std::for_each(vec.begin(), vec.end(), Printer());
```


Lambda Functions

```
struct Printer // our custom functor for console output
{
    void operator()(const string & str)
    {
        cout << str << endl;
    }
};

std::vector<string> vec = { "STL", "function", "objects", "rule" };

std::for_each(vec.begin(), vec.end(), Printer());

// using a lambda
std::for_each(vec.begin(), vec.end(),
              [](const string & str) { cout << str << endl; });
```

Lambda Functions

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

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

```
[ capture-list ] ( params ) { body }
```

```
[ capture-list ] { body }
```

Capture list can be passed as follows :

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

Anatomy of A Lambda

Lambdas == Functors

[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>

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 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**

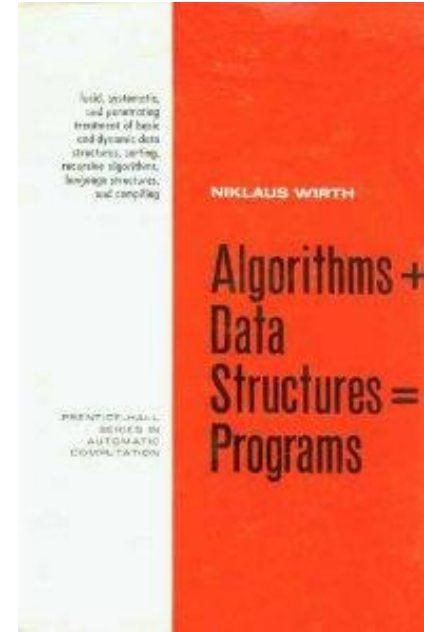
Why do we care ?

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

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**

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	$\forall a, \text{comp}(a, a) == \text{false}$
Antisymmetry	$\forall a, b, \text{if } \text{comp}(a, b) == \text{true} \Rightarrow \text{comp}(b, a) == \text{false}$
Transitivity	$\forall a, b, c, \text{if } \text{comp}(a, b) == \text{true} \text{ and } \text{comp}(b, c) == \text{true} \Rightarrow \text{comp}(a, c) == \text{true}$

{ 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 strcmp(s1.c_str(), s2.c_str()) < 0;
});
```

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:  $\frac{1}{2}$  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) )  Why do we need to check the value we searched for ?
{
    // 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);
```

