STL Algorithms
Principles and Practice

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

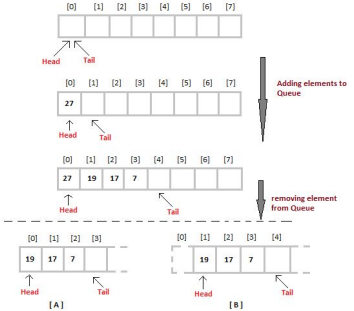
www.advancedinstaller.com

Winter 2017

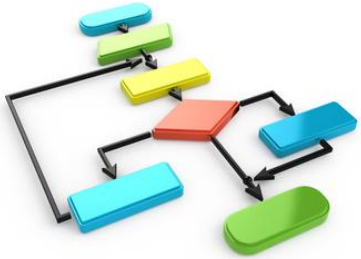
Agenda

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 Algorithms - Principles and Practice

(Part 2)

“Show me the code”

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 **`std::list`**

- `.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

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.

Prefer Member Functions To Similarly Named Algorithms

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

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


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

Why ?

Don’t Trust Your Intuition: Always `Benchmark` !

```cpp
static void StdFind(benchmark::State & state)
{
  std::set<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.insert("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;

  }
}

BENCHMARK(StdFind);
```

```cpp
static void SetFind(benchmark::State & state)
{
  std::set<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.insert("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = items.find("STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;
  }
}

BENCHMARK(SetFind);
```

http://quick-bench.com/d0kczl59jc0_4Mh7Gz_yKrs0-0E

```cpp
static void ListFind(benchmark::State & state)
{
  std::list<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.push_back("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;

  }
}

BENCHMARK(ListFind);
```

```cpp
static void VectorFind(benchmark::State & state)
{
  std::vector<std::string> items;
  for (int i = COUNT_ELEM; i >= 0; --i)
    items.push_back("string #" + std::to_string(i));

  // Code before the loop is not measured
  for (auto _ : state)
  {
    auto it = std::find(items.begin(), items.end(), "STL");
    if (it != items.end())
      std::cout << "Found: " << *it << std::endl;
  }
}

BENCHMARK(VectorFind);
```

http://quick-bench.com/U2yyY7YBqg3nsrzDIo_UIGANjPE

Try increasing values for `COUNT_ELEM` : 500 >>> 500’000 >>> ...

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);
}
```

*Why do we need to check the **value** we searched for ?*

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);
```

Fun with STL algorithms: What does it print ?

Homework

```
🚗 🛠️ = "algorithms";
🚗 🔗 = " ";
🚗 💔 = "really love";
🚗 🎶 = "!";

🚗 🏹(💰<📜> & 📦)
{
  🔢(📦.👉, 📦.👈, 🔱(🚗 & 💚, 🚗 & 💜)
  {
    return 💚.📏 < 💜.📏;
  });

  return 👰(📦.👉, 📦.👈, 📜(),
            🔱(🚗 & 🥹🥹, 🚗 & 🙄)
            {
              return (🥹🥹.🏳️ ? 🙄 : (🥹🥹 + 🔗)) + 🙄;
            });
}

int main()
{
  💰<📜> 😀😀😀 = { 🛠️, 💔, 🎶 };
  std::cout << 🏹(😀😀😀) << std::endl;
  return 0;
}
```

```
#include <iostream>
#include <string>
#include <algorithm>
#include <numeric>
#include <vector>

#define 🚗   const auto
#define 👰   std::accumulate
#define 🔢   std::sort
#define 🏳️   empty()
#define 📏   size()
#define 👉   begin()
#define 👈   end()
#define 🔱   []

using 📜 = std::string;
template<typename T>
using 💰 = std::vector<T>;
```

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 {   // our <algorithm_range.h> has ~150 wrappers for std algorithms

  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

Eg.

```
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(); } );
```

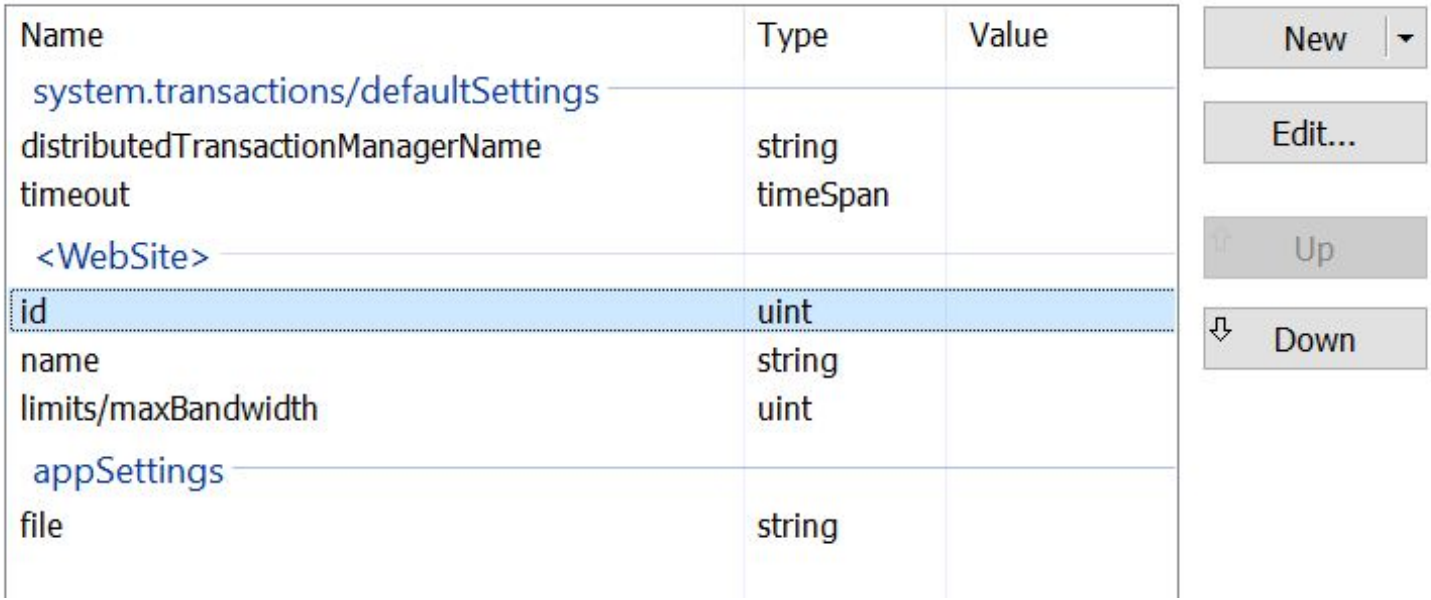
Calculating total number of unread messages.

```
// 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. Calculating total number of unread messages.

```
// 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;
                                    });
    });
}
```

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



C++ 14

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();
}
```

min

max

C++ 17

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

  if (aMoveUp)
    return min->GetOrderNumber() < aCurrentRow.GetOrderNumber();
  else
    return max->GetOrderNumber() > aCurrentRow.GetOrderNumber();
}
```

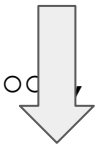
structured binding

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();

set<string> 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]));
```



Homework

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 }`

Homework

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
{
  // implement this
}
```

Student solutions for Homeworks

Homework 1 : `IterateSecond()` adapter

- Denis Ehorovici

Homework 2 : STL Snake game

- Denis Ehorovici
- Victor Ungureanu
- Ruxandra Lutan
- Ristea Stefan
- Andrei Popescu

Homework 3 : Emoji Algorithms

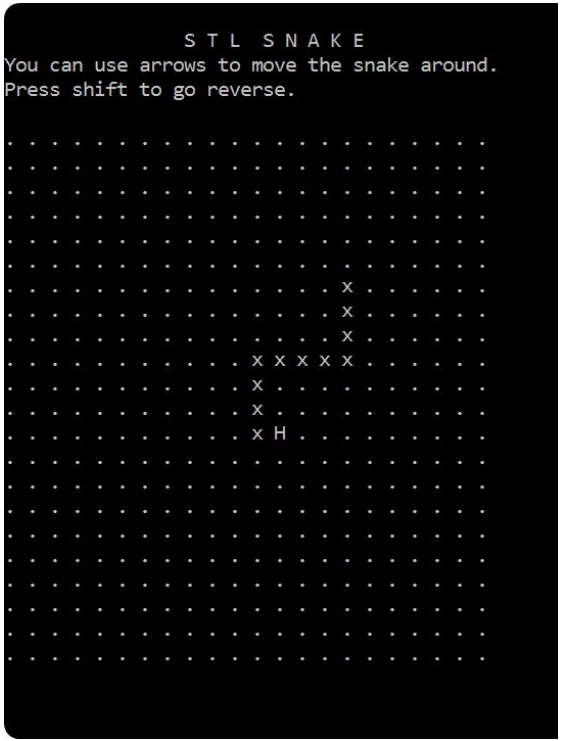
- TBA

Homework 4 : Server Nodes

- TBA

Demo: Time for coding fun!

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



Open with Visual Studio 2015/2017

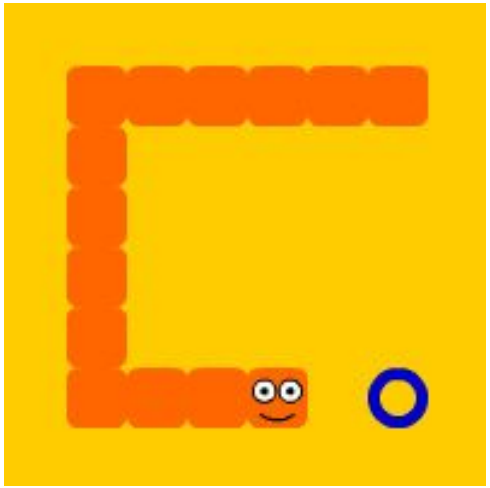
Search for **#STL** blocks

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

Is the snake still snakin' & dyin' right?

Demo: STL Snake

```
// Code walk-through
```



STL for Competitive Programming and Software Development

Coding Test

January 10, 2018
4pm

- 1 problem CAPHYON
- 1 problem NETROM
- aprox 3h
- open-books, internet
- bring your laptop

Course Evaluation:

"STL Algorithms - Principles and Practice" by CAPHYON

Winter 2017

Please take the survey:

https://www.surveymonkey.com/r/dcti2017

CAPHYON