The C++ Standard Template Library: Algorithms

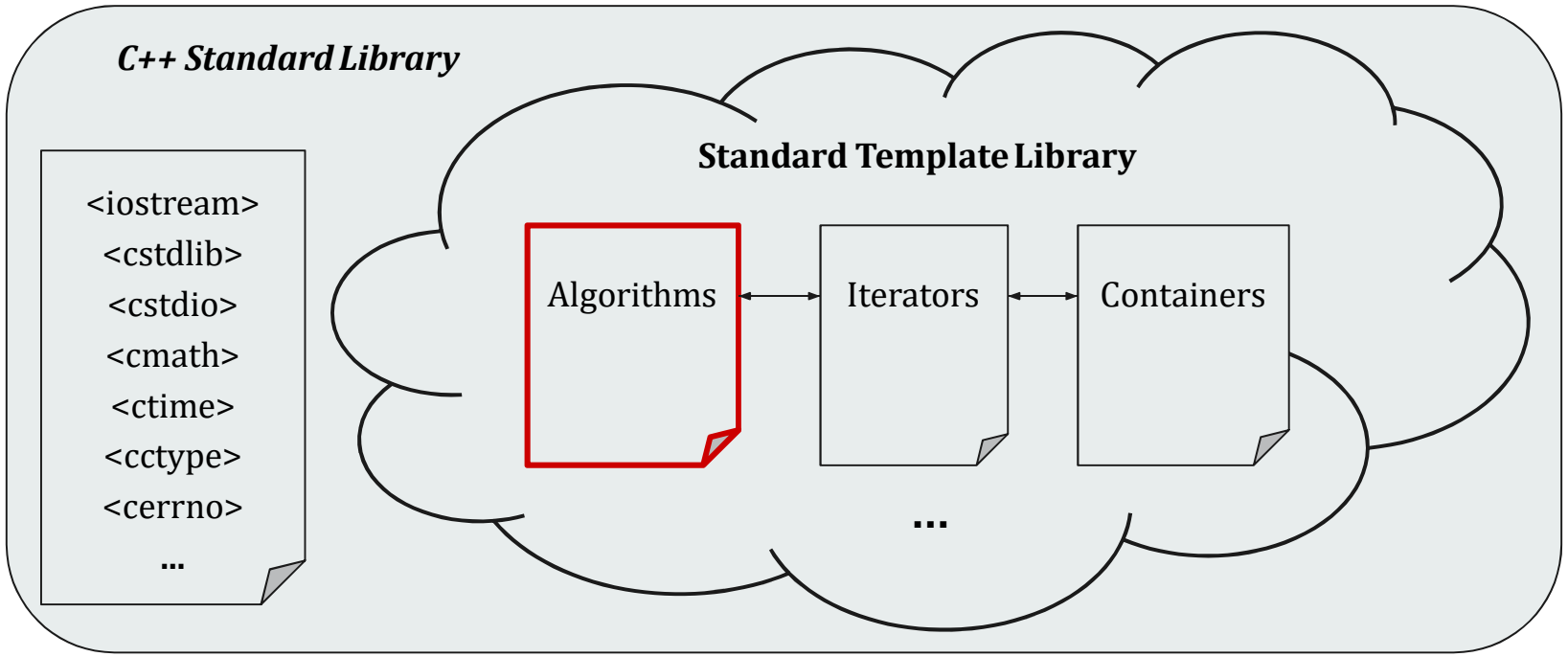
Concepts, usage, and best practices

Author: Pavlos Sakoglou, M.Sc.
pavlos@superharmonic.tech / github.com/PavlosSakoglou

Outline

- **Brief review of STL**
- STL Algorithms overview
- Elementary STL algorithms
- Bugs and Pitfalls
- Performance in development
- References
- Q&A

Brief review of STL



Brief review of STL

- ***STL Containers***: collection of most useful data structures
  - Sequence Containers
    - std::**vector**, std::**list**, std::**forward_list**, std::**deque**, std::**array**
  - Container Adaptors
    - std::**queue**, std::**stack**, std::**priority_queue**
  - Ordered/Unordered Associative Containers
    - std::**map**, std::**multimap**, std::**set**, std::**multiset**, std::**unordered_map**, std::**unordered_multimap**, std::**unordered_set**, std::**unordered_multiset**

More STL containers here: *http://en.cppreference.com/w/cpp/container*

Brief review of STL

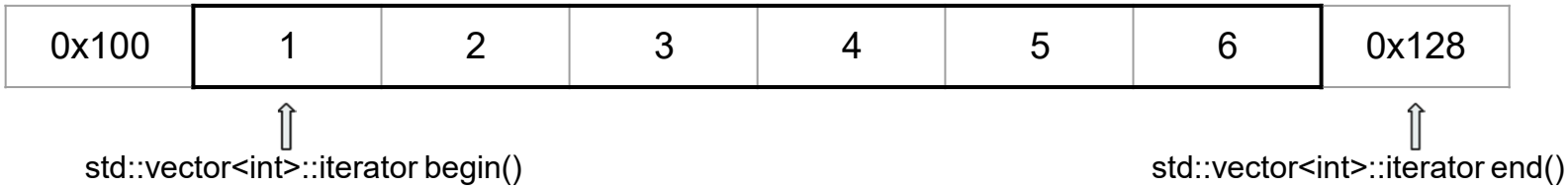
- ***STL Iterators:*** overloaded pointer interface for STL containers
  - ***Random access:*** access elements at any arbitrary offset position
  - ***Bidirectional:*** access elements by moving in both directions
  - ***Forward:*** access elements by only moving forward
  - ***Contiguous*** (C++17)***:*** like random access but logically adjacent elements also adjacent in memory

More iterators: *http://en.cppreference.com/w/cpp/iterator*

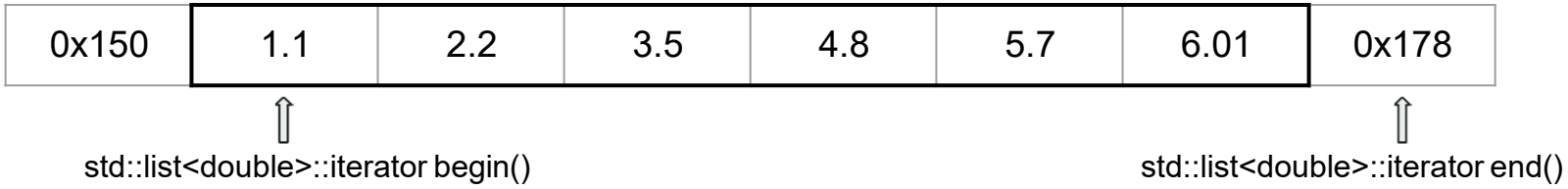
Brief review of STL

➢ ***STL Iterators***

std::vector<int> myVector = {1, 2, 3, 4, 5, 6};



std::list<double> myList = {1.1, 2.2, 3.5, 4.8, 5.7, 6.01};



Outline

- Brief review of STL
- **STL Algorithms overview**
- Elementary STL algorithms
- Bugs and Pitfalls
- Performance in development
- References
- Q&A

STL Algorithms overview

- What are STL algorithms?
  - Pre-built library of general purpose algorithms designed to solve specific problems i.e. sorting, searching, finding subsets, etc.
  - Free and standardized, used by thousands of developers daily
  - Declarative on syntax i.e. eliminate the need to write explicit loops
  - Iterate over some or all members of a *sequenced* STL container performing an operation on each element in turn
  - Designed by experts in numerical analysis and mathematics

STL Algorithms overview

- Why use STL algorithms?
  - Often more efficient than our own algorithms
  - Can leverage on the mechanics of STL containers
  - Cleaner code and clearly abstracted for several containers
  - Possible side effects contained inside the algorithm interface
  - Less likely to fail under non-obvious conditions
  - Intuitive identifiers, names, and interface
  - No need to reinvent the wheel

STL Algorithms overview

- ***STL Algorithms (Part 1)***
  - Non-modifying sequence operations
    - std::**for_each**, std::**count_if**, std::**search**, std::**find_if**, std::**equal, ...**
  - Modifying sequence operations
    - std::**copy_if**, std::**reverse**, std::**replace_if,** std::**transform,** ...
  - Sorting operations
    - std::**sort**, std::**is_sorted**, std::**partial_sort, ...**
  - Numeric operations
    - std::**accumulate**, std::**inner_product**, std::**partial_sum**, std::**iota, ...**

STL Algorithms overview

- ***STL Algorithms (Part 2)***
  - Set operations
    - std::**merge**, std::**includes**, std::**set_union**, std::**inplace_merge, ...**
  - Max/Min operations
    - std::**max_element,** std::**min_element**, std::**minimax_element, ...**
  - Heap operations
    - std::**is_heap**, std::**make_heap**, std::**push_heap**, std::**sort_heap, ...**

More STL algorithms: *http://en.cppreference.com/w/cpp/algorithm*

Outline

- Brief review of STL
- STL Algorithms overview
- **Elementary STL algorithms**
- Bugs and Pitfalls
- Performance in development
- References
- Q&A

Elementary STL algorithms

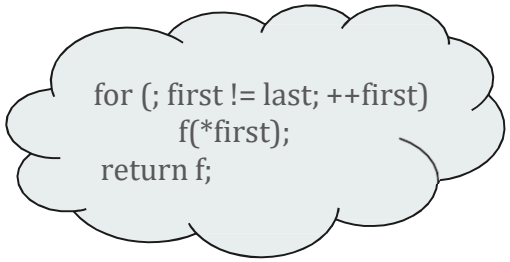
- std::**for_each**
- Defined in <algorithm> *http://en.cppreference.com/w/cpp/algorithm/for_each*

```
template <class InputIter, class UnaryFunction>
UnaryFunction for_each(InputIter first, InputIter last, UnaryFunction f) {

    for (; first != last; ++first)
        f(*first);
    return f;
}

// Input: a range given by two iterators, and a policy functor to apply on that range
// Output: policy functor or void
// Complexity: if policy f ~ O( f(n) ), then (last - first) * O( f(n) ) i.e. exactly (last - first) linear applications of f
```

Elementary STL algorithms

➢ std::**for_each** example: print all even numbers in a container

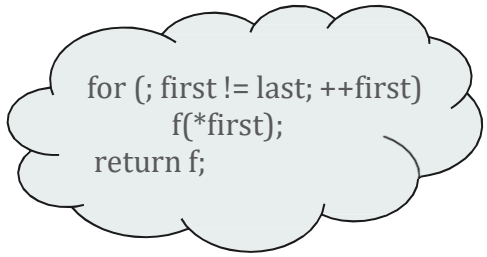
```
// “print if even” policy
struct print_if_even {
    void operator()(const int & number) {
        if (number % 2 == 0) std::cout << number << " ";
    }
};

// …
std::vector<int> myVector = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Print all even numbers of vector
std::for_each(myVector.begin(), myVector.end(), print_if_even());

// Will apply the policy to each element and result to:
// { print_if_even()(1), print_if_even()(2), ..., print_if_even()(10) } = { 1, print(2), 3, print(4), ..., 9, print(10) }
```



Elementary STL algorithms

➢ std::**for_each** example: “print if even” with offset

```
// print all even values of starting at the middle of the vector
// …
std::vector<int> myVector = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Get the middle and end of container using const iterators
std::vector<int>::const_iterator middle = myVector.begin() + myVector.size() / 2;
std::vector<int>::const_iterator end = myVector.end();

auto printing_evens = std::for_each(middle, end, print_if_even());

// What is ‘auto’? What is the type of ‘printing_evens’?
// What will the following line do?
// int k = 100; printing_evens(k);
```



Elementary STL algorithms

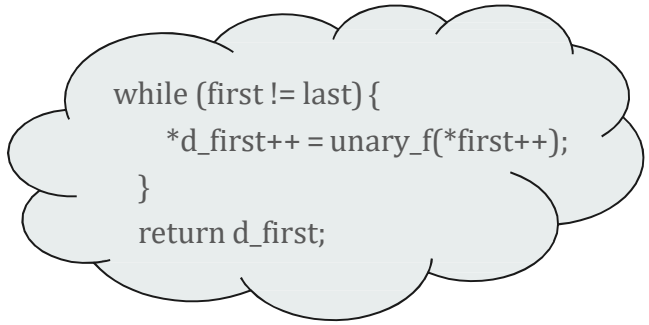
- std::**transform**
- Defined in <algorithm> *http://en.cppreference.com/w/cpp/algorithm/transform*

```
template<class InputIter, class OutputIter, class UnaryFunction>
OutputIter transform(InputIter first, InputIter last, OutputIter d_first, UnaryFunction unary_f) {
  while (first != last) {
    *d_first++ = unary_f(*first++);
  }
  return d_first;
}

// Input: a range of a container of values, an output iterator to assign the transformed values, and an operation
// Output: output iterator + (last - first) i.e. output container's end()
// Complexity: if unary_f ~ O(f(n)) then (last - first) * O(f(n)) i.e. exactly last - first applications of unary_f
```

Elementary STL algorithms

➢ std::**transform** example: exponentiate an array

```
template <class NumericType>
struct exponentiate {
    NumericType operator()(NumericType & element)
        { return std::exp(element); }
};

// ...
std::array<double, 5> myArray = { 3.4, 2.5, 9,8, 12.01 };

// Transform the current array
std::transform(myArray.begin(), myArray.end(), myArray.begin(), exponentiate<double>());
```



Elementary STL algorithms

- std::**copy_if**
- Defined in <algorithm> *http://en.cppreference.com/w/cpp/algorithm/copy*

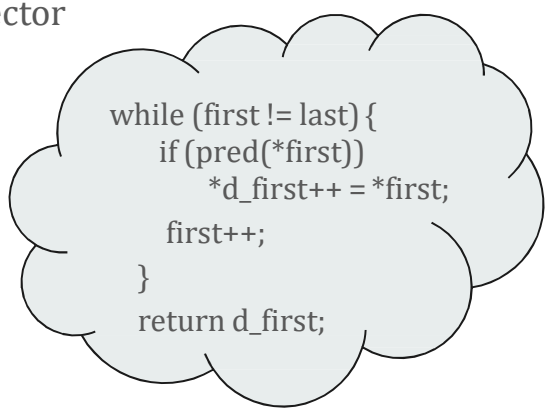
```
template <class InputIter, class OutputIter, class UnaryPredicate>
OutputIter copy_if(InputIter first, InputIter last, OutputIter d_first, UnaryPredicate pred) {
   while (first != last) {
        if (pred(*first)) *d_first++ = *first;
        first++;
   }
   return d_first;
}
```

```
// Input: a range of values to be copied, a start iterator to use for copying, and the predicate
// Output: the begin iterator of the new range
// Complexity: Linear O(last - first) i.e. exactly last - first assignment operator calls
```

Elementary STL algorithms

➢ std::**copy_if** example: copy all negative numbers to a vector

```
template <class NumericType>
struct is_negative {
public:
    bool operator()(const NumericType & value) { return value < 0; }
};

// …
std::list<double> myList = { -1.1, 2.231, -3.23, 4.01, -9.1, -89.1, 12, 8.28 };

std::vector<double> negatives;

std::copy_if(myList.begin(), myList.end(), std::back_inserter(negatives),  is_negative<double>());

// What is std::back_inserter? http://en.cppreference.com/w/cpp/iterator/back_inserter
// copy_if does not do a push_back, only copies the value on the position where the iterator is
```



Elementary STL algorithms

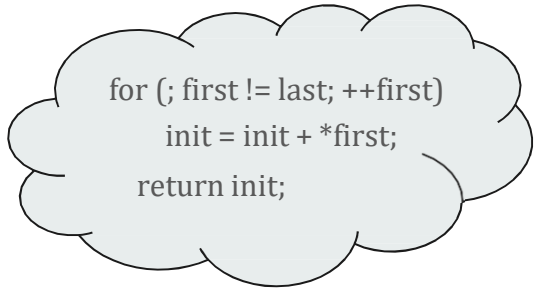
- std::**accumulate**
- Defined in <numeric> *http://en.cppreference.com/w/cpp/algorithm/accumulate*

```
template<class InputIter, class T>
T accumulate(InputIter first, InputIter last, T init) {

   for (; first != last; ++first)
      init = init + *first; // init is an offset
   return init;
}

// Input: a range of a container which values you want to accumulate (sum) and an initial value
// Output: the sum of values of the input range + the initial offset
// Complexity: linear O(last - first) i.e. exactly last - first addition operations
```

Elementary STL algorithms

➢ std::**accumulate** example: compute weekly investment earnings

```
double initial_earnings = 5123.98;

// Weekly earnings
std::array<double, 7> earnings = {10.3, -12.01, -8.9, 5.1, -7.8, 12.0, 1.1};

// Compute the new earnings by accumulating the daily values
double new_earnings =
    std::accumulate(earnings.begin(), earnings.end(), initial_earnings);

if (new_earnings > initial_earnings)
    std::cout << “Good job!\n”;
else
    std::cout << “We had losses!\n”;

initial_earnings = new_earnings;
```



Outline

- Brief review of STL
- STL Algorithms overview
- Elementary STL algorithms
- **Bugs and Pitfalls**
- Performance in development
- References
- Q&A

Bugs and Pitfalls

- Need to know involved STL features to use algorithms e.g. iterators
- You are not avoiding using pointers and their “buggy” behavior
- Newest algorithms won’t work with out-of-date compilers
- Doesn’t support native C++ code, only STL
- Easy to trigger run-time errors
- Not bug- or thread- safe

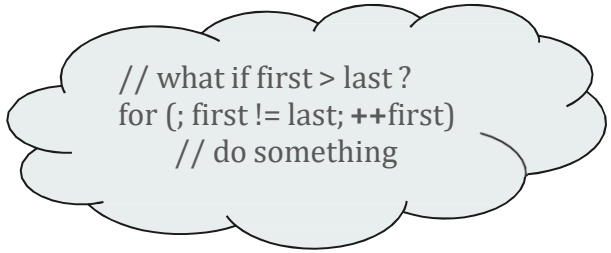
Bugs and Pitfalls

- Most common bugs when using STL algorithms:
  - Out-of-range input/output container iterators
  - Dereferencing *illegal* iterators implicitly
  - Poor container choices per algorithm
  - Container iterator const-correctness
  - Ignoring the algorithm's return type

Bugs and Pitfalls

➢ Out-of-range input/output iterators

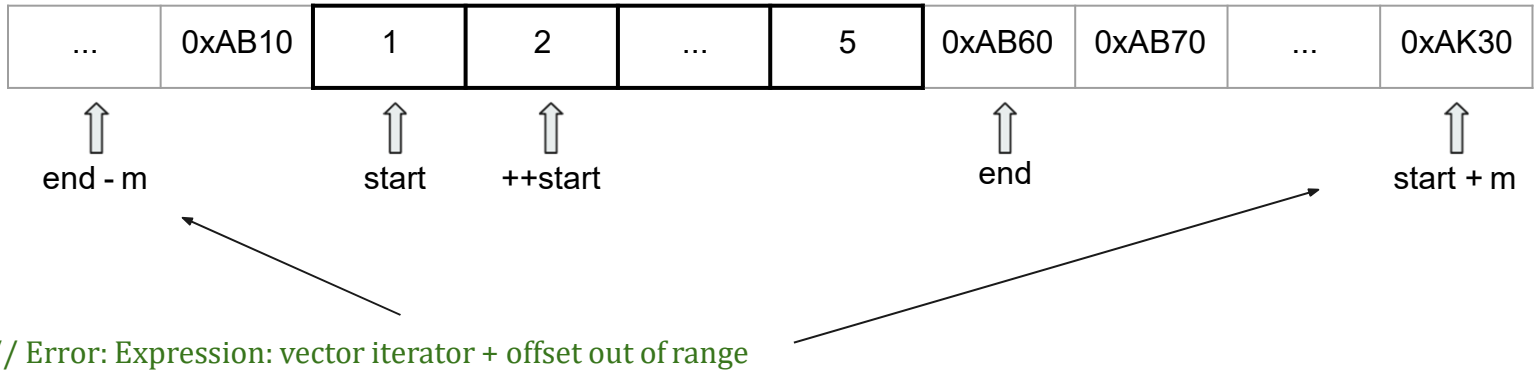
```
std::vector<int> myVector = {1, 2, 3, 4, 5};
std::size_t m = myVector.size() + k; // (int) k > 0

std::for_each(myVector.begin(), myVector.end() - m, policy<int>()); // yikes!
std::accumulate(myVector.begin() + m, myVector.end(), 0); // ouch!

// Error: Expression: vector iterator + offset out of range


std::accumulate(myVector.end(), myVector.begin(), 0); // crash!

// Error: Expression: invalid iterator range
```



Bugs and Pitfalls

➢ Out-of-range input/output iterators

```
std::vector<int>::iterator start = myVector.begin();
std::vector<int>::iterator end = myVector.end();
int m = size + k; // (int) k > 0
```



Bugs and Pitfalls

- Dereferencing *illegal* iterators

```
while (first != last) {
    if (pred(*first))
        *d_first++ = *first;
    first++;
}
return d_first;
```

```
// Implicit illegal dereference
// …
std::list<double> myList2 = { -1.1, 2.231, -3.23, 4.01, -9.1, -89.1, 12, 8.28 };
std::vector<double> negatives;

// The following line will throw an exception:
//      std::copy_if(myList2.begin(), myList2.end(), negatives.begin(), is_negative<double>());

// first = myList2.begin();
// last = myList2.end();
// d_first = negatives.begin(); // negatives is empty !
// ...
// *d_first++ = *first; // BUG: dereferencing uninitialized iterator !
// …
// Error message: vector iterator not incrementable (why?)
```

Bugs and Pitfalls

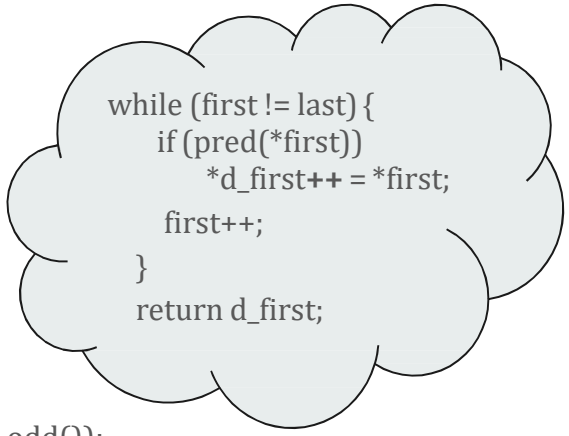
➢ Poor container choices and const-correctness

```
// std::set

std::set<int> mySet = { 2, 3, 5, 7, 9 };
// std::reverse(mySet.begin(), mySet.end()); // Compiler error !

// C3892: https://msdn.microsoft.com/en-us/library/cab7058b.aspx
// std::set is inherently constant. You can add and remove elements, but not modify/permute them

// std::unordered_map

std::unordered_map<int, int> myHash = { {1,1}, {2,2}, {3,3} };
// std::sort(myHash.begin(), myHash.end()); // Compiler errors !

// No comparison operator for std::unordered_map -- could not deduce type etc.
```

Bugs and Pitfalls

- Ignoring the return type

```
// Explicit illegal dereference

std::array<int, 10> intArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::array<int, 5> oddArray;

std::array<int, 5U>::iterator d_first =
    std::copy_if(intArray.begin(), intArray.end(), oddArray.begin(), is_odd());

// After the last *d_first++ = *first assignment, the ++ operator returns and
// increments d_first to one position past the last element

// std::cout << *d_first << std::endl; // ouch!

// Error: Expression: array iterator not dereferencable
```



Outline

- Brief review of STL
- STL Algorithms overview
- Elementary STL algorithms
- Bugs and Pitfalls
- **Performance in development**
- References
- Q&A

Performance in development

➢ Problem 1: sort a vector of chars in increasing and decreasing order (part 1)

// Native C++ solution on github: *https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1/blob/master/Problem1_Native.cpp*

```
template <class Type>
int partition(std::vector<Type> & myVector, int begin, int end) {/*...*/}

template <class Type>
void myQuicksort(std::vector<Type> & myVector, int begin, int end) {/*...*/}

// Sort
myQuicksort<char>(myVector, 0, size - 1); // O(n lg n)

// Copy values to increasing-order vector
for (const double & elem : myVector)
	increasing.push_back(elem);


// Reverse copy
auto begin = myVector.begin(); auto end = myVector.end(); int i = 1;

for (; begin != end; ++begin) decreasing.push_back(*(end - i++));
```

Performance in development

➢ Problem 1: sort a vector of chars in increasing and decreasing order

// STL solution on github: *https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1/blob/master/Problem1_STL.cpp*

```
// STL Sort
std::sort(myVector.begin(), myVector.end()); // O(n lg n)


// Copy increasing
std::copy(myVector.begin(), myVector.end(), std::back_inserter(increasing));  // O(n)


// Copy decreasing
std::reverse_copy(myVector.begin(), myVector.end(), std::back_inserter(decreasing));  // O(n)
```

Performance in development

➢ Problem 2: computing basic and nth order statistics (part 1)

// Native C++ solution on github: *https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1/blob/master/Problem2_Native.cpp*

```
// ...
template <class Type>
void myQuicksort(std::vector<Type> & myVector, int begin, int end) {/*...*/}

// Average
for (const double & elem : prices)
	daily_average += elem;
daily_average /= (double)size;

// Median
myQuicksort(prices, 0, size - 1);
daily_median = prices[size / 2 - 1];

// Copy max 5 prices
for (unsigned i = 0; i < 5; ++i)
	max_5_prices[i] = prices[size - 1 - i];
```

Performance in development

➢ Problem 2: computing basic and nth order statistics (part 2)

// Native C++ solution on github: *https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1/blob/master/Problem2_Native.cpp*

```
// Max - Min prices
auto first = std::begin(prices);
auto last = std::end(prices);


if (first != last)
	min_price = max_price = *first++;


for (const double & elem : prices) {

	if (elem > max_price)
		max_price = elem;

	if (elem < min_price)
		min_price = elem;
}
```

Performance in development

➢ Problem 2: computing basic and nth order statistics

// STL solution on github: *https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1/blob/master/Problem2_STL.cpp*

```
// Average
daily_average = std::accumulate(std::begin(prices), std::end(prices), 0.0) / (double)size;


// Median
std::partial_sort(std::begin(prices), std::begin(prices) + (size / 2) + 1, std::end(prices));
daily_median = prices[size / 2 - 1];

// Max - Min price
auto max_price_position = std::max_element(std::begin(prices), std::end(prices));
if (max_price_position != prices.end()) max_price = *max_price_position;

auto min_price_position = std::min_element(std::begin(prices), std::end(prices));
if (min_price_position != std::end(prices)) min_price = *min_price_position;

// Max 5 prices
std::nth_element(std::begin(prices), std::begin(prices) + 5, std::end(prices), decreasing_order<double>());
std::copy(std::begin(prices), std::begin(prices) + 5, std::back_inserter(max_5_prices));
```

Performance in development

➢ Problem 2: computing basic and nth order statistics

```
// BONUS: compute variance using for_each and a lambda expression vs C++11 range-based for loop

double variance = 0.0;

// STL
std::for_each(std::begin(prices), std::end(prices), [&](double elem) {
        variance += std::pow(elem - daily_average, 2);
});

// Native C++
for (const double & elem : prices)
        variance += std::pow(elem - daily_average, 2);

// Compute the unbiased variance
variance /= (double)(size - 1);


// std::for_each or range-based for-loop ?
```

Outline

- Brief review of STL
- STL Algorithms overview
- Elementary STL algorithms
- Bugs and Pitfalls
- Performance in development
- **References**
- Q&A

References

- Thorough online documentation:
  - http://en.cppreference.com/w/cpp/algorithm
  - http://www.cplusplus.com/reference/algorithm/
  - http://www.sgi.com/tech/stl/
- Suggested open-source texts:
  - http://stepanovpapers.com/STL/DOC.PDF
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf
- Presentation repository with all examples and codes:
  - https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1

References

- What's next?
  - Advanced STL algorithms and iterators
  - C++ Lambda functions and STL function objects
  - Augmenting STL data structures and algorithms
  - STL algorithms and memory management

That is all!

The C++ Standard Template Library: Algorithms

Concepts, usage, and best practices

Thank you for your time!

Author: Pavlos Sakoglou, M.Sc.
pavlos@superharmonic.tech / github.com/PavlosSakoglou