

The C++ Standard Template Library: Algorithms

Concepts, usage, and best practices

Lecture 1 notes

January 3rd, 2017



Pavlos Sakoglou, M.Sc.

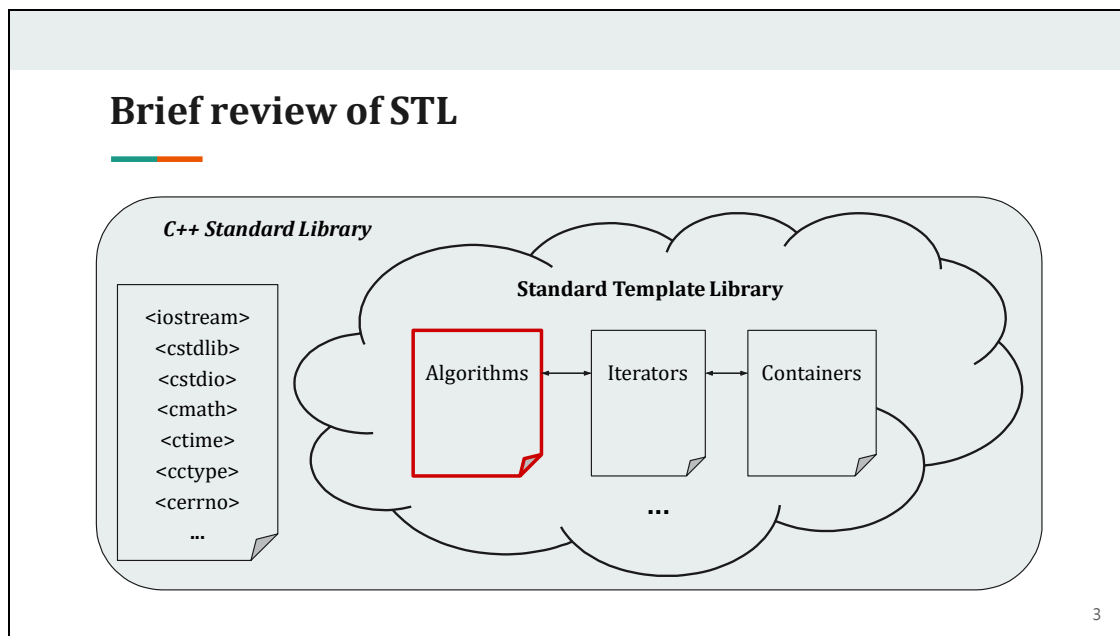
pavlos@superharmonic.tech / <https://github.com/PavlosSakoglou>

Contents

1. Brief review of STL
2. STL Algorithms overview
3. Elementary STL algorithms
4. Bugs and Pitfalls
5. Performance in development
6. Conclusion & References

Brief review of STL

The Standard Template Library (STL) is a subset of the C++ Standard Library and provides generic functionality that enables a developer to collect and operate on data in a simple and efficient way. STL was built based on the idea of the generic programming paradigm, that is, a programming method that allows functionality to be written in terms of types to-be-defined later. As a result, with the use of templates we can write generic code that will be overloaded later to multiple types, but has only one definition template.



The two most basic STL libraries we need in our study of STL Algorithms are the Container and Iterator libraries. STL Containers are generic classes that encapsulate and augment traditional data structures like static and dynamic arrays, linked lists, binary trees, hash tables, queues, stacks, and more.

In contrast to the Object-Oriented paradigm where we encapsulate data and operations on data in classes, STL Algorithms are based on the functional programming paradigm. That is, we separate data from algorithms, and we use iterators as intermediate objects between them. The typical case is to iterate a generic STL container from beginning to end, and apply an operation to each element separately.

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>

4

Most STL Algorithms refer to sequence containers, like arrays, vectors, double-ended queues (deques), and singly and doubly linked lists. Occasionally, we can use STL Algorithms to associative containers like sets and maps, however, due to their nature, associative containers have their own functionality built-in in their structure. For example, a set is a representation of a binary tree and thus all its operations take logarithmic time to execute. Thus, it would be pointless to apply an STL search algorithm that takes linear time for an STL set that provides its own logarithmic-order *find* method. The same holds with adaptor containers. Adaptor containers are using sequence container interface but with further restrictions. For example, a stack uses a dynamic array, like the vector class, but with the restriction that you can only use the elements at the end of the container, with *push-*, *pop-*, and *top-* like methods. As a result, it will be meaningless to attempt to use an STL sorting algorithm for a stack.

Iterators are overloaded pointer interfaces that are provided for each container and allow the programmer to efficiently use them to access and manipulate elements, as per the data structure. The most commonly used iterators are the Random-Access Iterators that can be incremented and decremented in both directions, but also to move forward and backward with step *n* at a time, emulating the traditional array pointers that can be dereferenced and return an element at any position in constant time. Bidirectional Iterators can only move forward and backward with step 1, emulating a doubly linked list pointer that cannot jump to a few elements forward, but only arrive there sequentially.

Similar, but more limited, are the Forward Iterators, that mimic singly linked list pointers, and consequently can only be incremented in one direction by step 1.

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>

5

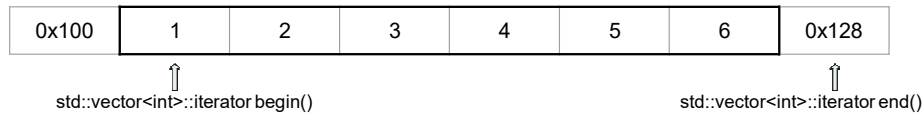
Iterators can be dereferenced and modify the value of the element of the STL Container they are pointing to, if they are not constant (read-only) iterators. Keep in mind that every STL Container with a type has a unique iterator that cannot be used interchangeably with an iterator of a different type or container. In other words, an iterator of a vector of integers is different than an iterator of a vector of doubles, let alone an iterator of a list of integers.

Like pointers, iterators can be tricky to use. Make sure that an iterator points to a legal Container address before you use or dereference it. A common bug occurs when programmers fail to consider that the end iterator of a container points to the memory cell one past the last element of the container.

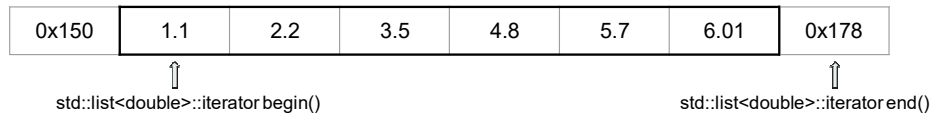
Brief review of STL

➤ 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};
```



6

STL Algorithms overview

An algorithm is a procedure to solve a problem in a finite number of well-defined steps. A sequential algorithm performs operations one at a time. This contrasts with distributed or parallel algorithms that can perform many steps simultaneously by distributing their execution among several cores or processors.

The Standard Template Algorithms Library was originally developed in the late 1970s and became part of the C++ Standard in the late 1990s. This means that every C++ compiler supports STL and its free to use in development by everyone. In addition, STL algorithms are optimized, which means that the C++ compilers have minimized the memory requirements and the size of the executable generated by STL code. This makes development with STL more efficient in general, but that's not always the case as we will see below.

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

8

Additionally, STL algorithms have intuitive and clean syntax which makes the code more readable, and were written by experts in mathematics and numerical analysis. We can trust that STL provides fast and optimized functionality for our programs, makes our code easy-to-read and -maintain.

Although STL Algorithms are not perfect and we can still trigger exceptions in our code by using them, they are bug-safe when used with proper handling. Unless we make an obvious mistake with the input or output variables of the algorithm, it's very unlikely that the algorithm won't work.

The biggest advantage that the STL Algorithms provide is the fact that a programmer does not have to re-invent the wheel. This means that for standardized procedures in our code, like searching or sorting, we can completely rely on STL rather our own hand-written algorithms. For example, even though we can write a great sorting algorithm (which can be very expensive to bullet-proof and test) it is unlikely that we will gain in performance versus using STL's sort function. Thus, STL helps us to avoid writing predefined processes and that makes the development process much faster and safer.

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

9

There are eleven STL Algorithm categories overall, but in this introductory lecture we will focus and see example from seven of them. Once we understand the mechanics of the following categories, by induction, it will be easy to work with the rest, since all of them are based on the same principle:

1. Non-modifying sequence operations

- `std::for_each`, `std::count`, `std::count_if`, `std::find`, `std::equal`, etc.

These algorithms operate on a range of elements without modifying the values. Their purpose is to use the values as input to further conditions and function pointers

2. Modifying sequence operations

- `std::copy_if`, `std::copy`, `std::reverse`, `std::swap`, `std::replace_if`, `std::transform`, etc.

These algorithms operate on a range of elements by applying a modification rule on each element, optionally copying the modified values to another STL Container

3. Sorting operations

- `std::sort`, `std::is_sorted`, `std::partial_sort`, etc.

These algorithms implement several comparison sorting algorithms (quicksort, merge sort, etc.) and are used to sort a range of elements of an STL Container

4. Numeric operations

- `std::accumulate`, `std::inner_product`, `std::partial_sum`, `std::iota`, etc.

These algorithms perform numeric operations on a range of numeric values or types that overload appropriately numeric operators. Mostly used for arithmetic operations

5. Set operations

- `std::merge`, `std::includes`, `std::set_union`, `std::merge`, etc.

These algorithms perform the common set operations (union, intersection, etc.) on sorted sequence STL Containers

6. Max/Min operations

- `std::max_element`, `std::min_element`, `std::min`, `std::max`, `std::minimax_element`, etc.

These algorithms are used to find the maximum and minimum element between two elements, or in a range of elements

7. Heap operations

- `std::is_heap`, `std::make_heap`, `std::push_heap`, `std::sort_heap`, etc.

These algorithms implement heap-like operations on a range of elements, such as building a max heap out of an array of elements, heapifying a new element, checking if the range is already a heap or not, etc.

There are cases where we can combine a non-modifying algorithm with a callable object that modifies the sequence. We need to observe that STL won't give any flags about such operations, and that as programmers we are responsible to keep best practices. Nevertheless, oftentimes such conceptual violations perform better in our programs. In the next section we will examine some elementary examples.

Elementary STL algorithms

The first algorithm we are going to discuss is `std::for_each`. It is defined in the `<algorithm>` header and like all other STL functionality, is part of the namespace `std`.

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  $f \sim O(f(n))$ , then  $(last - first) * O(f(n))$  i.e. exactly (last - first) linear applications of f
```

13

As you can see in the slide, `for_each` has a linear complexity and operates on all elements sequentially of the input iterator range. It takes a begin and an end iterator of an STL sequence container, and a function object as input, and applies the function object to each element. Thus, its complexity is exactly linearly many operations of the function object.

Elementary STL algorithms

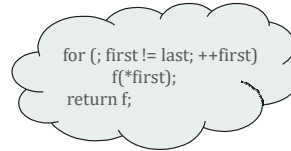
➤ `std::for_each` example: print all even numbers in a container

```
// "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) }
```



14

In the example above, the container we are using is an `std::vector` which supports a Random-Access iterator. As a result, we can choose an arbitrary offset in the begin and end iterator, and customize the range of elements of the container we want to apply the input function object on.

Elementary STL algorithms

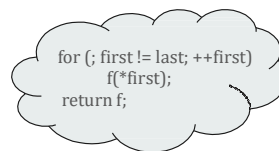
➤ `std::for_each` example: "print if even" with offset

```
// 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);
```



15

A similar (but slightly less efficient) algorithm is `std::transform`. Like `for_each`, the `std::transform` algorithm iterates on the input range and applies the input policy function on each element, but additionally it creates a copy of each element to an output container iterator, which is one of the parameters. You can optionally choose to copy the values to a new container or the same container, thus modifying the existing structure. This is what makes it less efficient, as you could instead have used `for_each` and save a linear amount of copies.

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
```

16

The complexity of `std::transform` is linearly many applications of the input function object (policy). One of the advantages of `std::transform` is that it allows you to copy elements from different types of containers i.e. transform and copy elements from `std::list` to `std::vector`. This extra functionality with a small cost in performance makes our code more flexible.

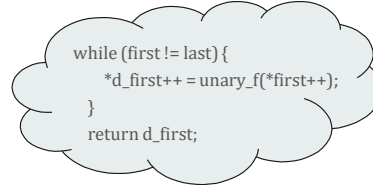
Elementary STL algorithms

- `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>());
```



17

The following algorithm we are examining is `std::copy_if`. As the name suggest, this algorithm iterates in a range of elements of a container, and given an input policy function, decides whether each element will be copied to another container, that is also given as input. If the input policy evaluates true, the element gets copied to the output iterator, which then gets incremented for the next potential element.

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
```

18

In the example below, we want to copy all negative values from a doubly linked list, to a vector. Thus, we define a function object policy to be used as predicate condition, we iterate on the input range of the linked list, and when an element satisfies the predicate i.e. when an element is negative, it gets copied (pushed back) to the output vector of negatives.

Elementary STL algorithms

➤ `std::copy_if` example: copy all negative numbers to a vector

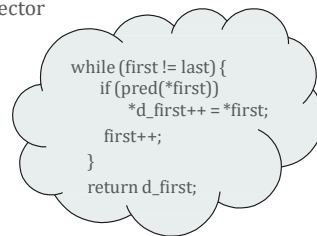
```
template<class NumericType>
struct is_negative {
public:
    bool operator()(const NumericType & value) { return value < 0; }
};

// ...
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
```



Although this process seems straightforward, it's important to notice that `copy_if`, and other similar algorithms, should not be used to modify the elements inside the predicate function that is applied. The reason is separation of concerns. It's bad practice to fuse functionality of functions of different purpose. A predicate function is used to return true or false as per a pre-defined condition. It's not expected to modify the element in the process. Other programmers will take it for granted that it keeps everything constant. Make sure that your implementation of your own function objects (predicates or not) will be consistent with their definition to avoid confusions in a shared development project.

The `std::find` algorithm takes a range of elements and a value of same type as input, and compares every element sequentially to the input value. The first element that is equal to the input value stops the iteration and the algorithm returns the iterator to that element. If no element matches the input value, then the algorithm returns the end iterator of the input range.

Elementary STL algorithms

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

```
template<class InputIter, class T>
InputIter find(InputIter first, InputIter last, const T& target) {
    for (; first != last; ++first)
        if (*first == target) // dereference and compare
            return first;
    return last;
}

// Input: a range given by two iterators, and a value to compare with
// Output: iterator to the first element with value that matches the target, or the end() iterator if no value matches
// Complexity: O(last - first) i.e. at most linear in last - first range
```

20

A typical usage would be to look for the first occurrence of a value. Make sure that the value is found before you use it, otherwise you will get an exception for dereferencing the *end* iterator.

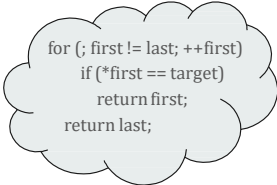
Elementary STL algorithms

- `std::find` example: increment the first occurrence of element with value 2

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

std::vector<int>::iterator found = std::find(myVector.begin(), myVector.end(), 2);

if (found != myVector.end())
    *found++; // 2 becomes 3
else
    std::cout << "Element not found!\n";
```



```
for (; first != last; ++first)
    if (*first == target)
        return first;
return last;
```

21

The `std::accumulate` algorithm is defined in the header `<numeric>` and its purpose is to iterate over a range of elements, sum up the values sequentially, and return their sum. Additionally to the input range, `std::accumulate` takes an initial value that is adding the elements to. This is particularly useful in cases where we need to compute a sum with an initial value. If we are only interested in the sum of the range, we set the initial value to zero.

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
```

22

In the example below, we want to compute the weekly earnings from an investment. We consider the initial value of `std::accumulate` to be the portfolio value at the end of the last week, and we add to it the daily earnings of the past week (seven days). Eventually, we check whether we had profit or loss and we update the portfolio value with the new earnings.

The alternative of `std::accumulate` would be `for_each` with a policy, or a range-based C++11 `for` loop. Although the politically correct way is to use `std::accumulate` for numeric sums, we need to reckon that `for_each` with a slight modification would be more flexible in summing up non-numeric types, if the addition operator was properly overloaded for that type.

Elementary STL algorithms

- `std::accumulate` example: compute weekly investment earnings

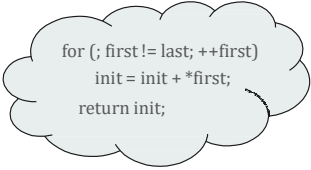
```
double initial_earnings = 5123.98;

// 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;
```



```
for (; first != last; ++first)
    init = init + *first;
return init;
```

23

Bugs and Pitfalls

To state the obvious: STL is not perfect. In this section we are going to investigate some of the disadvantages that come with the Algorithms library, the most common bugs that occur in software development, and ways to tackle them.

Below is a (non-linked 😊) list of some disadvantages:

- To use STL Algorithms, one needs to have a solid understanding of STL Containers, STL Iterators, and preferably knowledge of a few more advanced concepts, such as lambda functions, STL binders, and STL function objects and wrappers. Although these concepts are not strict requirements to use Algorithms, one needs to know them to fully leverage the use of STL Algorithms.

- Since STL Algorithms can only operate on STL Containers, it forces the developer to use STL in general for every software component that includes an STL algorithm. That might lead to bloated production code, since the use of templates can indirectly slow down the program. Although, the obvious delays due templates are at compile time, the use of templates can lead to slow execution by simply producing enough object code that does not fit on the cache. For example, in native C++ code the compiler will take the initiative to convert `int*` to `int const*` and as a result function calls that use either `const` and non-`const` arguments will have the same code (unless you explicitly overload `const` and non-`const` versions of the function). With templates that won't happen, and the compiler will need two entirely separate pieces of code being generated.
- The use of STL Algorithms do not avoid the use of pointers – masked as iterators. One needs to be careful when dereferencing a returned output iterator, or when passes an iterator range as an argument with an offset. It is strongly suggested to be careful with iterators as if they were raw pointers. As a result, STL algorithms can trigger a lot of run-time errors if they are not used responsibly.
- STL Algorithms are not thread-safe. You should not expect them to be bullet-proof and production ready. The developers need to treat STL as a great and correct, generic implementation of an algorithm, and provide any safety measures themselves as if they are using native C++ code.

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

Other issues might appear when using STL Algorithms. The most common bugs and errors are the following:

➤ Out-of-range input/output iterators

This mistake causes a run-time errors, such as “out of range iterators”. It occurs in cases that the begin iterator is in a greater position in memory than the end iterator, and thus the implicit iteration will never come to an end.

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
```

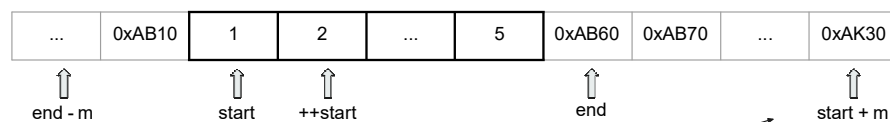
// what if first > last ?
for (; first != last; ++first)
// do something

27

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
```



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

28

➤ Dereferencing illegal iterators implicitly

This error happens when you ignore the mechanics of an algorithm, i.e. when you pass an output iterator of an uninitialized range that points to an uninitialized address, and gets dereferenced.

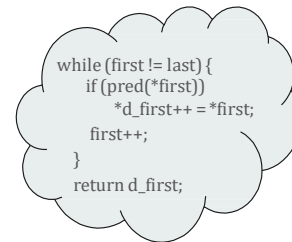
Bugs and Pitfalls

➤ Dereferencing *illegal* iterators

```
// 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?)
```



29

➤ Poor container choices per algorithm and const-correctness

When you are attempting modifying operations on a constant-valued container or a container that does not support comparison operations.

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.
```

30

➤ Ignoring the algorithm's return type

Whenever the returned output iterator points to an invalid address and the programmer fails to consider that case.

Bugs and Pitfalls

➤ Ignoring the return type

// Explicit illegal dereference

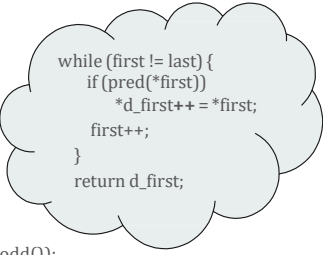
```
std::array<int, 10> intArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
std::array<int, 5> oddArray;
```

```
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



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

31

Performance in development

In this section we are going to consider two problems and compare two solutions for each, one in STL and another in native C++ code.

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

Problem 2: compute basic and nth order statistics on a set of stock prices

In both solution we will use an auxiliary customized function to populate the vector with characters at random:

```
// O(input_size) (linear)
// Initialize the vector to random characters of the English alphabet
void myCharInit(std::vector<char> & myVec) { /* ... */ }
```

Problem 1: Native C++ solution

```
// Include Standard Library headers
#include <iostream>
#include <vector>
#include <ctime>
#include <random>

// Partition auxiliary function for quicksort
template <class Type>
int partition(std::vector<Type> & myVector, int begin, int end) {
    Type x = myVector[end];
    int i = begin - 1;
    for (unsigned j = begin; j < end; j++) {
        if (myVector[j] <= x) {
            i++;
            Type temp = myVector[j];
            myVector[j] = myVector[i];
            myVector[i] = temp;
        }
    }
    Type temp = myVector[i + 1];
    myVector[i + 1] = myVector[end];
    myVector[end] = temp;
    return i + 1;
}

// Recursive Quicksort algorithm
template <class Type>
void myQuicksort(std::vector<Type> & myVector, int begin, int end) {
    if (begin < end) {
        int mid = partition(myVector, begin, end);
        myQuicksort(myVector, begin, mid - 1);
        myQuicksort(myVector, mid + 1, end);
    }
}
```



```

int main() {

    // 1. Create a vector populated with characters

    const std::size_t size = 10;

    std::vector<char> myVector(size);
    myCharInit(myVector);

    // 2. Sort
    myQuicksort<char>(myVector, 0, size - 1); // O(n lg n)

    // 3. Copy values to increasing-order vector

    std::vector<char> increasing;

    for (const double & elem : myVector)
        increasing.push_back(elem);

    // 4. Copy values to decreasing-order vector

    auto begin = myVector.begin();
    auto end = myVector.end();

    int i = 1;

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

    return 0;
}

```

This works. However, we had to implement a version of quicksort algorithm and a reverse function. In a good case scenario our implementation will be efficient and bug-free, but this costs us time and energy. Notice that the purpose of this problem was to get a vector of characters as input and to output two variant vectors of the input, one in increasing order and the other in decreasing order. Instead we spent a couple hours implementing and testing quicksort thoroughly. This is extra cost that can be avoided with STL Algorithms.

Problem 1: STL solution

```
// Include Standard Library headers
#include <iostream>
#include <algorithm>
#include <vector>
#include <random>
#include <chrono>
#include <iterator>

int main() {

    // Create a vector populated with characters
    const std::size_t SIZE = 20;
    std::vector<char> myVector(SIZE);
    myCharInit(myVector);

    // Result vectors
    std::vector<char> increasing, decreasing;

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

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

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

    return 0;
}
```

This solution is more elegant and more compact without any extra time or space cost. This saves us development time and we are guaranteed that the STL Algorithms will do their job. Additionally, the procedure here is more intuitive. The STL identifiers are very clear and another developer, less familiar with STL can still understand what this code is doing without much trouble. Now our code is more readable.

Problem 2: Native C++ Solution

```
// Include Standard Library headers
#include <iostream>
#include <string>
#include <vector>
#include <utility> // std::pair

// O(n lg n) Quicksort

// Partition auxiliary function for quicksort
template <class Type>
int partition(std::vector<Type> & myVector, int begin, int end) { /* ... */}

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

int main() {

    // 1. Get the prices

    // Daily stock prices, updated every 30 minutes
    // std::pair
    // http://en.cppreference.com/w/cpp/utility/pair

    std::vector<std::pair<std::string, double>> input_prices =

    { { "09:30AM", 23.29 }, { "10:00AM", 22.11 }, { "10:30AM", 23.42 }, { "11:00AM", 23.64 },
      { "11:30AM", 22.95 }, { "12:00PM", 22.81 }, { "12:30PM", 22.98 }, { "01:00PM", 24.65 },
      { "01:30PM", 25.10 }, { "02:00PM", 25.12 }, { "02:30PM", 25.96 }, { "03:00PM", 24.98 },
      { "03:30PM", 24.65 }, { "04:00PM", 23.45 } };

    // 2. Process the prices

    // Get a non-zero size
    std::size_t size;
    (input_prices.size() != 0) ? size = input_prices.size() : size = 1;

    // Convert values to a vector
    std::vector<double> prices;

    // C++ ranged-based for loop
    // Get the numeric values i.e.
    // elem.first = key
    // elem.second = value

    for (auto & elem : input_prices)
        prices.push_back(elem.second);
```

```

// 3. Compute statistics

    // i. Average price

double daily_average = 0.0;

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

daily_average /= (double)size;


    // ii. Median price

double daily_median = 0.0;

myQuicksort(prices, 0, size - 1);

daily_median = prices[size / 2 - 1];


    // iii. Max price

double max_price = 0.0;

auto first = std::begin(prices);
auto last = std::end(prices);

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

for (const double & elem : prices)
    if (elem > max_price)
        max_price = elem;


// Since 'prices' is a sorted vector from computing the median,
// we could instead do the following:
// max_price = prices[size - 1]; // last element is the largest


    // iv. Min price

double min_price = 0.0;

first = prices.begin();
last = prices.end();

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

for (const double & elem : prices)
    if (elem < min_price)
        min_price = elem;


// Since 'prices' is a sorted vector from computing the median,
// we could instead do the following:
// max_price = prices[0]; // first element is the smallest

```

```

        // v. Price range
double price_range = 0.0;
price_range = std::abs(max_price - min_price);

        // v. Max 5 prices
std::vector<double> max_5_prices(5);

// Since 'prices' is a sorted vector from computing the median,
// we can do the following:
for (unsigned i = 0; i < 5; ++i)
    max_5_prices[i] = prices[size - 1 - i];

// Alternatively, we would have to sort the vector first and then copy the values

        // vi. Variance
double variance = 0.0;

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

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

return 0;
}

```

As you might have noticed, there are many loops and many repetitive operations. Additionally, we had to re-create a version of quicksort to help us determine the median, which as we saw in the previous example, it can be expensive.

Below we see an alternative, that does not necessarily improve performance, but makes the code more compact, more readable, bug-free, and easier to maintain.

Problem 2: STL solution

```
// Include Standard Library headers
#include <iostream>
#include <string>
#include <numeric>
#include <algorithm>
#include <vector>
#include <iterator>
#include <utility> // pair

// Policy (callable object / functor)

// Use this predicate for comparison sorting condition
// Will sort in decreasing order

template <class Numeric>
struct decreasing_order {
    bool operator()(const Numeric & a, const Numeric & b)
    { return a > b; }
};

int main() {

    // 1. Get and 2. Process the prices
    // ...

    // 3. Compute statistics

        // i. Average price

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

        // ii. Median price

        double daily_median = 0.0;
        std::partial_sort(std::begin(prices),
                        std::begin(prices) + (size / 2) + 1, std::end(prices));

        daily_median = prices[size / 2 - 1];

        // iii. Max price

        double max_price = 0.0;
        std::vector<double>::iterator max_price_position =
            std::max_element(std::begin(prices), std::end(prices));

        if (max_price_position != prices.end()) max_price = *max_price_position;
```

```

        // iv. Min price

double min_price = 0.0;
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;


        // v. Price range

double price_range = 0.0;
price_range = std::abs(max_price - min_price);


        // v. Max 5 prices

std::vector<double> 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));


        // vi. Variance (with C++11 lambda expression)

double variance = 0.0;
std::for_each(std::begin(prices), std::end(prices), [&](double elem) {
    variance += std::pow(elem - daily_average, 2);
});

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

return 0;
}

```

Most of the operations now require only one line of code. Our solution with STL is cleaner, took less development time, and has same (and potentially better) performance.

Conclusion & References

STL is a useful and sharp tool to use in development. It's free, intuitive, optimized, and if used responsibly will decrease the likelihood of having bugs in our code.

STL's major advantage is that a developer does not have to re-invent the wheel. We can refer to STL Algorithms for standardized operations in our code, and as a result produce more compact and more clean solutions without losing focus off the main objective of the software we are designing.

Some basic disadvantages are that STL Algorithms can only work with STL containers, and that they require a good knowledge of more involved STL functionality, like STL Iterators and Function Objects. Additionally, if we over-use STL, it might make our executable slightly slower and eventually cost us in performance.

To avoid bugs, we need to be careful with the input and output iterators we use, and always keep track of what is getting dereferenced at which part of the algorithm. Referring to STL Algorithms documentation online is tremendously helpful and helps us understand the mechanics of the algorithms by providing us with possible implementations, return types, input arguments, complexity analysis, and elementary examples.

Below find reference links for further online resources and documentation on STL Algorithms:

<http://en.cppreference.com/w/cpp/algorithm>

<http://www.cplusplus.com/reference/algorithm/>

<http://www.sgi.com/tech/stl/>

<http://stepanovpapers.com/STL/DOC.PDF>

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf>

The presentation Github repository with all codes and examples can be found on this address:

<https://github.com/PavlosSakoglou/STL-Algorithms-Tutorial-1>

Happy learning!

