STL Algorithms in Action

STL Algorithms and their everyday application

Michael VanLoon
CPPcon 2015

What are algorithms?

al·go·rithm

noun: algorithm; *plural noun*: algorithms

A procedure for solving a … problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end especially by a computer.

Merriam-Webster dictionary

What are STL algorithms?

- A pre-built library of general-purpose algorithms designed to solve specific problems
- Come “for free” with your C++ compiler
- Operate on sequences or sequenced containers
- Declarative in syntax – no explicit (“raw”) loops
- Iterate over some or all members of a sequence performing an operation on each element in turn
- Designed by experts and proven bug-free by the millions of lines of other peoples’ programs that already use them!

What is a raw loop anyway?

- It's a `for`, `while`, or `do… while` loop
- Explicitly coded
- Often contains many lines of code (should it?)
- May cause side-effects outside its scope

```
vector<int> out;
bool found = false;
for (const auto& i: v) {
    if (i >= 42) {
        out.emplace_back(i);
        ++global_count;
        if (i == 42) {
            found = true;
        }
    }
}
```

Why use algorithms?

- Often more efficient than hand-written loop
- Cleaner and more clearly abstracted than a raw loop
- Contains side-effects inside a clear interface
- Prevents accidental leakage of side-effects
- Eases reasoning about functionality and reasoning about post conditions
- Less likely to fail under non-obvious conditions
- Eases reasoning about the surrounding code

Classes of STL algorithms

- Non-modifying sequence operations (25.2)
- Mutating sequence operations (25.3)
- Sorting and related operations (25.4)
- General C algorithms (25.5)
- General numeric operations (26.7)

(section of the C++ standard INCITS/ISO/IEC 14882-2011[2012])

STL algorithms

non-modifying sequence operations

- Do not modify the input sequence.
- Do not emit a result sequence.
- *Algorithm* will not cause side-effects in input sequence.
- *Function object*, if present, may cause side-effects by modifying itself, the sequence (in certain cases, e.g. `for_each`), or its environment.

STL algorithms

non-modifying sequence operations

- all_of
- any_of
- none_of
- for_each
- find
  - find_if
  - find_if_not
- find_end
- find_first_of
- adjacent_find
- count
  - count_if
- mismatch
- equal
- is_permutation
- search
  - search_n

STL algorithms

mutating sequence operations

- Do not modify the input sequence, except in situations when output overlaps input, resulting in modification in-place (e.g. `transform`).
- Emit an output sequence of results.
- Output sequence may potentially overlap input sequence for certain algorithms (e.g. `transform`). Others (e.g. `copy`) explicitly disallow overlap/in-place.
- *Algorithm* will explicitly cause side-effects in output sequence.
- *Function object*, if present, may cause side-effects by modifying itself or its environment. *Function object* should not modify the input or output sequences.

STL algorithms
mutating sequence operations

- copy
  - copy_n
  - copy_if
  - copy_backward
- move
  - move_backward
- swap_ranges
  - iter_swap
- transform
- replace
  - replace_if
  - replace_copy
  - replace_copy_if
- fill
  - fill_n
- generate
  - generate_n
- remove
  - remove_if
  - remove_copy
  - remove_copy_if
- unique
  - unique_copy
- reverse
  - reverse_copy
- rotate
  - rotate_copy
- shuffle
  - random_shuffle
- partition
  - is_partitioned
  - stable_partition
  - partition_copy
  - partition_point

STL algorithms

sorting and related operations

- A mix of non-modifying and mutating operations
- Mutating operations modify sequences in place (e.g. `sort`, `make_heap`), or emit output to an output sequence (e.g. `merge`, `partial_sort_copy`)
- Default compare function is `operator<`
- Explicit compare function object, if supplied, must not modify the sequence or iterators

STL algorithms
sorting and related operations

- sorting
  - sort
  - stable_sort
  - partial_sort
  - partial_sort_copy
- nth_element
- binary search
  - lower_bound
  - upper_bound
  - equal_range
  - binary_search
- merge
  - merge
  - inplace_merge
- set operations on sorted structures
  - includes
  - set_union
  - set_intersection
  - set_difference
  - set_symmetric_difference
- heap operations
  - push_heap
  - pop_heap
  - make_heap
  - sort_heap
- minimum and maximum
  - min
  - max
  - minmax
  - min_element
  - max_element
  - minmax_element
- lexicographical comparisons
  - lexicographical_compare
- permutation generators
  - next_permutation
  - prev_permutation

STL algorithms
general numeric operations

- Library of algorithms for doing numeric operations.
- Consist of components for complex number types, random number generation, nu- meric (*n*-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library.[1]

[1]Description from the standard library that is surprisingly understandable by humans.

STL algorithms
general numeric operations

- `accumulate`
- `inner_product`
- `partial_sum`
- `adjacent_difference`
- `iota`

STL algorithms

C library algorithms

- These are shown for completeness.
- You may need to know about these for legacy reasons.
- In general, there is nothing these can do that you can’t do better with the modern algorithms previously mentioned.

STL algorithms
C library algorithms

- `bsearch`
- `qsort`

`for_each` **and** `transform`

- Your go-to generic algorithms for doing general things to sequences
- Applies an operation to each element in a sequence, in order
- Very similar except completely different

for_each

- Applies an operation to each element in a sequence (like many algorithms)
- Is a non-modifying sequence operation
- *Algorithm* produces no side-effect
- *Function object* may produce a side-effect by modifying the input sequence
- *Function object* may produce a side-effect by modifying itself
- Returns a moved copy of the function object
- `for_each` is considered non-modifying because it produces no output range; it relies on the function object for mutation, if any

`transform`

- Applies an operation to each element in a sequence (like many algorithms)
- Is a mutating sequence operation
- If the input range(s) and result range are the same, or overlap, mutates objects in-place
- *Algorithm* explicitly produces a side-effect
- *Function object* may not produce a side-effect
- `transform` is considered mutating because it explicitly produces an output range modified by applying the function object to elements, and forbids the function object from modifying any of the range elements or iterators
- Returns iterator pointing one past last element in result range

for_each example

Generate a single hash for all strings in a vector

```
struct HashString {
    void operator()(const string& s) {
        hash = accumulate(s.begin(), s.end(), hash, hash_char);
    }
    uint32_t hash = 0;
};
```

accumulate

- Is a non-modifying numerics operation
- *Algorithm* produces no side-effect
- *Function object* may not modify the sequence or the iterator
- *Function object* may produce a side-effect by returning a return code different from input parameter
- `accumulate` differs from `for_each` in that the algorithm carries a value rather than a function object from visit to visit, applying the operation to each element and the current value
- `accumulate` differs from `for_each` in that it has a default operation: `operator+`

for_each example

Generate a single hash for all strings in a vector

```
struct HashString {
    void operator()(const string& s) {
        hash = accumulate(s.begin(), s.end(), hash, hash_char);
    }
    uint32_t hash = 0;
};
```

for_each example

Generate a single hash for all strings in a vector

```
struct HashString {
    void operator()(const string& s) {
        hash = accumulate(s.begin(), s.end(), hash, hash_char);
    }
    uint32_t hash = 0;
};

template<typename Cont>
uint32_t hash_all_strings(const Cont& v) {
    const auto hasher = for_each(v.begin(), v.end(), HashString());
    return hasher.hash;
}
```

for_each example

Generate a single hash for all strings in a vector

```
struct HashString {
    void operator()(const string& s) {
        hash = accumulate(s.begin(), s.end(), hash, hash_char);
    }
    uint32_t hash = 0;
};

template<typename Cont>
uint32_t hash_all_strings(const Cont& v) {
    const auto hasher = for_each(v.begin(), v.end(), HashString());
    return hasher.hash;
}

void test_for_each_hash() {
    vector<string> v{ "one", "two", "three", "four", "five" };
    uint32_t hash = hash_all_strings(v);
    cout << "Hash: " << hash << dec << endl;
}
```

for_each example (cont)

... the rest of the code...

```
uint32_t rotl(uint32_t value, unsigned int count) {
    const uint32_t mask =
        (CHAR_BIT * sizeof(value) - 1);
    count &= mask;
    return (value << count) |
           (value >> ((-count)&mask));
}

uint32_t hash_char(uint32_t hash, char c)
{
    hash = rotl(hash, c); // circular rotate left
    hash ^= c;
    return hash;
}
```

transform example

Generate hash for each string in a vector

```
uint32_t hash_string(const string& s) {
    return accumulate(s.begin(), s.end(), 0, hash_char);
};
```

transform example

Generate hash for each string in a vector

```
uint32_t hash_string(const string& s) {
    return accumulate(s.begin(), s.end(), 0, hash_char);
};

template<typename Cont>
vector<uint32_t> hash_each_string(const Cont& v) {
    vector<uint32_t> res;
    transform(v.begin(), v.end(), back_inserter(res), hash_string);
    return res;
}
```

transform example

Generate hash for each string in a vector

```
uint32_t hash_string(const string& s) {
    return accumulate(s.begin(), s.end(), 0, hash_char);
};

template<typename Cont>
vector<uint32_t> hash_each_string(const Cont& v) {
    vector<uint32_t> res;
    transform(v.begin(), v.end(), back_inserter(res), hash_string);
    return res;
}

void test_transform_hash() {
    vector<string> v{ "one", "two", "three", "four", "five" };
    auto res = hash_each_string(v);
    cout << "Hashes: ";
    for_each(res.begin(), res.end(),
             [](uint32_t rh){ cout << rh << " "; });
    cout << endl;
}
```

any_of, all_of, and none_of

- Apply a function object to a sequence
- Determines whether any, all, or none of the elements in the sequence are true as determined by the function object
- May return before evaluating all elements in sequence if outcome is determined early

all_of example

validate http headers

```
 static const regex
     reHeader("([A-Za-z0-9!#$%&'*+.^_`|~-]+): *(.+) *");

inline bool
headers_valid(const vector<string>& headers) {
    return all_of(headers.begin(), headers.end(),
        [](const auto& header) -> bool {
            smatch matches;
            return regex_match(header, matches, reHeader);
        }
    );
}
```

all_of example

validate http headers test and output

```
void all_of_headers() {
    vector<string> h1 = { "Foo: bar", "Content-type: application/json",
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha << headers_valid(h1) << endl;

    vector<string> h2 = { "Foo : bar", "Content-type: application/json",
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha << headers_valid(h2) << endl;

    vector<string> h3 = { "Foo: bar", "Content-type: application/json",
                          ":Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha << headers_valid(h3) << endl;

    vector<string> h4 = { "Foo: bar", " Content-type: application/json"
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha << headers_valid(h4) << endl;
}
```

output:

```
headers valid: true
headers valid: false
headers valid: false
headers valid: false
```

any_of example

http header search

```
inline bool
header_search(const vector<string>& headers,
    const string& find_header, const string& find_value)
{
    return any_of(headers.begin(), headers.end(),
        [&find_header, &find_value](const auto& header) -> bool {
            const regex reHeader(
                "(" + find_header + "): *(" + find_value + ") *",
                regex::icase);
            smatch matches;
            return regex_match(header, matches, reHeader);
        }
    );
}
```

any_of example

http header search test and output

```
void any_of_headers_simple() {
    vector<string> h1 = { "Foo: bar", "Content-type: application/json",
                          "X-SuperPower: toestrength",
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha
         << header_search(h1, "X-SuperPower", "toestrength") << endl;

    vector<string> h2 = { "Foo: bar", "Content-type: application/json",
                          "X-SuperPower: supersmell",
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha
         << header_search(h2, "X-SuperPower", "toestrength") << endl;

    vector<string> h3 = { "Foo : bar", "Content-type: application/json",
                          "X-SuperPower: toestrength",
                          "Accept: text/html,text/json,application/json" };
    cout << "headers valid: " << boolalpha
         << header_search(h3, "X-Superpower", "toeStrength") << endl;
}
```

output:

```
headers valid: true
headers valid: false
headers valid: true
```

another for_each example

simultaneously validate and search http headers

```
struct HeaderData {
    int good_headers = 0;
    int bad_headers = 0;
    multimap<string, string> found_headers;

    string find_header;
    string find_value;

    operator bool() const { return !bad_headers && good_headers > 0; }

    void operator() (const string& header) {
        static const regex reValid("([A-Za-z0-9!#$%&'*+.^_`|~-]+): *(.+) *");
        smatch matches;
        bool match = regex_match(header, matches, reValid);
        if (match) {
            ++good_headers;
            const regex reHeader("(" + find_header + "): *(" + find_value + ") *", regex::icase);
            if (regex_match(header, matches, reHeader)) {
                found_headers.emplace(matches[1], matches[2]);
            }
        } else {
            ++bad_headers;
        }
    }
};
```

another for_each example

simultaneously validate and search http headers

```
struct HeaderData {
    int good_headers = 0;
    int bad_headers = 0;
    multimap<string, string> found_headers;

    string find_header;
    string find_value;

    operator bool() const;
    void operator() (const string& header);
};

const HeaderData header_parse(const vector<string>& headers, const string&
find_header, const string& find_value) {
    HeaderData hd;
    hd.find_header = find_header;
    hd.find_value = find_value;
    return for_each(headers.begin(), headers.end(), hd);
}
```

another for_each example

simultaneous validate/search test

```
void any_of_headers_full() {
    {
        vector<string> h1 = { "Foo: bar", "Content-type: application/json", "X-SuperPower: toestrength",
                              "Accept: text/html,text/json,application/json" };
        const HeaderData& hd = header_parse(h1, "X-SuperPower", "toestrength");
        cout << "headers parse: " << hd << ", good " << hd.good_headers << ", bad " << hd.bad_headers;
        for_each(hd.found_headers.begin(), hd.found_headers.end(), [](const auto& val) {
                cout << "\n\t'" << val.first << "', '" << val.second << "'";
            });
        cout << endl;
    }

    {
        vector<string> h2 = { "Foo: bar", "Content-type: application/json", "X-SuperPower: supersmell", "Accept: text/ht
ml,text/json,application/json" };
        const HeaderData& hd = header_parse(h2, "X-SuperPower", "toestrength");
        cout << "headers parse: " << hd << ", good " << hd.good_headers << ", bad " << hd.bad_headers;
        for_each(hd.found_headers.begin(), hd.found_headers.end(), [](const auto& val) {
                cout << "\n\t'" << val.first << "', '" << val.second << "'";
            });
        cout << endl;
    }

    {
        vector<string> h3 = { "Foo : bar", "Content-type: application/json", "X-Superpower: toestrength", "Accept: text/
html,text/json,application/json" };
        const HeaderData& hd = header_parse(h3, "X-SuperPower", "toestrength");
        cout << "headers parse: " << hd << ", good " << hd.good_headers << ", bad " << hd.bad_headers;
        for_each(hd.found_headers.begin(), hd.found_headers.end(), [](const auto& val) {
                cout << "\n\t'" << val.first << "', '" << val.second << "'";
            });
        cout << endl;
    }
}
```

another `for_each` example

simultaneous validate/search output

output:

```
headers parse: true, good 4, bad 0
      'X-SuperPower', 'toestrength'
headers parse: true, good 4, bad 0
headers parse: false, good 3, bad 1
      'X-Superpower', 'toestrength'
```

adjacent_find

- *adjacent_find* searches for adjacent items (pairs of elements next to each other) in a sequence that meet a certain condition.
- Returns an iterator to the first of the pair of elements meeting the condition.
- The default condition is equality (i.e. find two adjacent items that are equal).
- A custom comparator may be provided to look for other adjacent conditions.

adjacent_find example

simple is_sorted implementation

```
 vecInt_t v{ 1, 2, 3, 4, 5, 5, 6, 7, 8 };

// Greater works because it's asking if the first value is
// greater than the second value.  If so, then the test
// fails (not sorted). If the first value is less than or
// equal to the second value, no match and success.
vecInt_t::iterator it =
    adjacent_find(v.begin(), v.end(), greater<int>());

if (it == v.end())
    cout << "Vector is sorted" << endl;
else
    cout << "Vector not sorted, value " << *(it + 1)
         << ", at position " << it - v.begin() + 1 << endl;
```

output:

```
Vector is sorted
Vector not sorted, value 3, at position 9
```

adjacent_find example

test for sequence deviation

```
template<typename Cont>
typename Cont::const_iterator checkDeviation(const Cont& cont,
                                             double allowed_dev)
{
    return adjacent_find(cont.begin(), cont.end(),
                         [allowed_dev](const typename
                                       Cont::value_type& v1,
                                   const typename
                                       Cont::value_type& v2)
                         {
                             auto limit = v1 * allowed_dev;
                             return (v2 > v1 + limit) ||
                                    (v2 < v1 - limit);
                         });
}
```

adjacent_find example

test for sequence deviation test and output

```
vecDbl_t v{ 1.0, 1.05, 1.06, 1.04, 1.09, 1.15, 1.2 };

vecDbl_t::const_iterator it = checkDeviation(v, 0.1);
if (it == v.end())
    cout << "Vector is within deviation limits" << endl;
else
    cout << "Vector outside deviation limits, values " << *it << " and "
         << *(it + 1) << ", at position " << it - v.begin() + 1 << endl;

v.push_back(2.0);

it = checkDeviation(v, 0.1);
if (it == v.end())
    cout << "Vector is within deviation limits" << endl;
else
    cout << "Vector outside deviation limits, values " << *it << " and "
         << *(it + 1) << ", at position " << it - v.begin() + 1 << endl;
```

output:

```
Vector is within deviation limits
Vector outside deviation limits, values 1.2 and 2, at position 7
```

`remove_if` (with `erase`)

Scott Meyers, "Effective STL," items 9 and 32

- Scenario: you want to erase several items from a container that meet a condition
- You could write a loop with some checks, some explicit erases, and potential iterator invalidation
- Or…

remove_if (with erase)

Scott Meyers, "Effective STL," items 9 and 32

```
struct Foo {
    bool expired;
    … other members…
};

vector<Foo> v;

v.erase(
    remove_if(v.begin(), v.end(),
              [](const Foo& foo){ return foo.expired; }),
    v.end());
```

Output:

```
before: [val: 1, expired: false] [val: 2, expired: true] [val: 3,
expired: false] [val: 4, expired: false] [val: 5, expired: true]
after:  [val: 1, expired: false] [val: 3, expired: false] [val: 4,
expired: false]
```

Know your sorts and sorta-sorts

Scott Meyers' "Effective STL" Item 31

- Sorting algorithms:
  - `sort`[1]
  - `stable_sort`[1]
  - `partial_sort, partial_sort_copy`[1]
- Sorta-sorts:
  - `nth_element`[1]
  - `partition, partition_copy`[2]
  - `stable_partition`[2]

[1] Requires random access iterators

[2] Requires bidirectional iterators

Know your sorts

Scott Meyers' "Effective STL" Item 31

- `sort`
  - Most general-purpose sort
  - Order of equivalent items implementation-defined
  - In some cases, may be more efficient than `stable_sort` since equivalent items can be rearranged at sort's discretion
  - Sorts in place
- `stable_sort`
  - Order of equivalent items preserved
  - Sorts in place
- `partial_sort`
  - Sort a subset of a sequence, drawing from a subset that is equal to or larger than the sorted sequence
  - There is no stable version of `partial_sort`
- `partial_sort_copy`
  - Like `partial_sort`, but…
  - Sorts specified subset of an input sequence, emitting to an output sequence

sort and stable_sort

sort objects by last name, first, middle

- Scenario: assume an object with strings containing fist name, middle name, and last name, among other things

```
struct Person {
    string first;
    string middle;
    string last;
    … other Person stuff…
};
```

- We want to sort said objects by all three fields, with precedence last > first > middle

sort and stable_sort

sort objects by last name, first, middle

```
vector<Person> v{{
        { "Joe", "P", "Smith" },
        { "Jane", "Q", "Jones" },
        { "Frank", "P", "Johnson" },
        { "Sarah", "B", "Smith" },
        { "Joe", "X", "Jones" },
        { "Joe", "A", "Smith" } }};

// Sort by least influential data first
sort(v.begin(), v.end(),
     [](const Person& a, const Person& b){ return a.middle < b.middle; });
```

sort and stable_sort

sort objects by last name, first, middle

```
vector<Person> v{{
        { "Joe", "P", "Smith" },
        { "Jane", "Q", "Jones" },
        { "Frank", "P", "Johnson" },
        { "Sarah", "B", "Smith" },
        { "Joe", "X", "Jones" },
        { "Joe", "A", "Smith" } }};

// Sort by least influential data first
sort(v.begin(), v.end(),
     [](const Person& a, const Person& b){ return a.middle < b.middle; });

stable_sort(v.begin(), v.end(),
            [](const Person& a, const Person& b){ return a.first < b.first; });
```

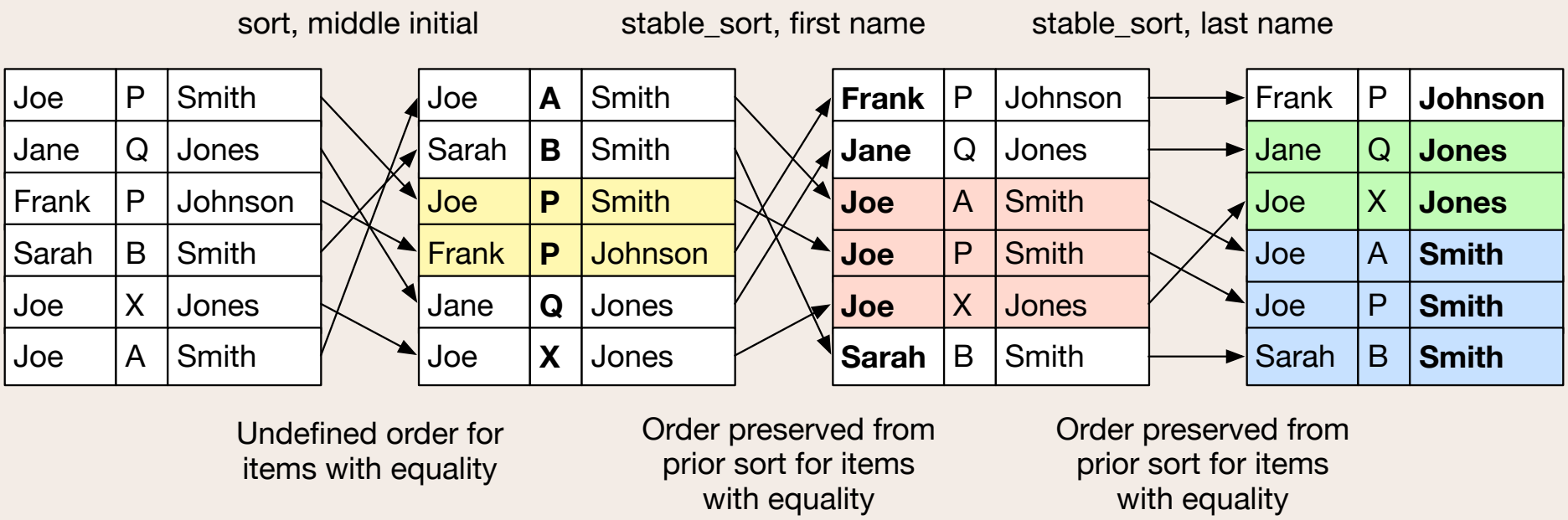
sort and stable_sort

sort objects by last name, first, middle

```
vector<Person> v{{
        { "Joe", "P", "Smith" },
        { "Jane", "Q", "Jones" },
        { "Frank", "P", "Johnson" },
        { "Sarah", "B", "Smith" },
        { "Joe", "X", "Jones" },
        { "Joe", "A", "Smith" } }};

// Sort by least influential data first
sort(v.begin(), v.end(),
     [](const Person& a, const Person& b){ return a.middle < b.middle; });

stable_sort(v.begin(), v.end(),
            [](const Person& a, const Person& b){ return a.first < b.first; });

stable_sort(v.begin(), v.end(),
            [](const Person& a, const Person& b){ return a.last < b.last; });
// Sort by most influential data last
```

sort and stable_sort

visual: sort objects by last name, first, middle



partial_sort

- Takes an input sequence of sort candidates
- Sorts the top n elements into a potentially smaller output sequence
- Order of items from input sequence that are unsorted in output sequence are in implementation defined order
- `partial_sort` is an in-place operation
- `partial_sort_copy` copies sorted output to a separate sequence
- `partial_sort` is obviously more efficient than a full sort

partial_sort example

```
vector<int> v1{ 42, 17, 89, 22,
                34, 78, 63, 12,
                57, 99 };

partial_sort(v1.begin(),
             v1.begin() + 5,
             v1.begin() + 8,
             greater<int>());
```

partial_sort example

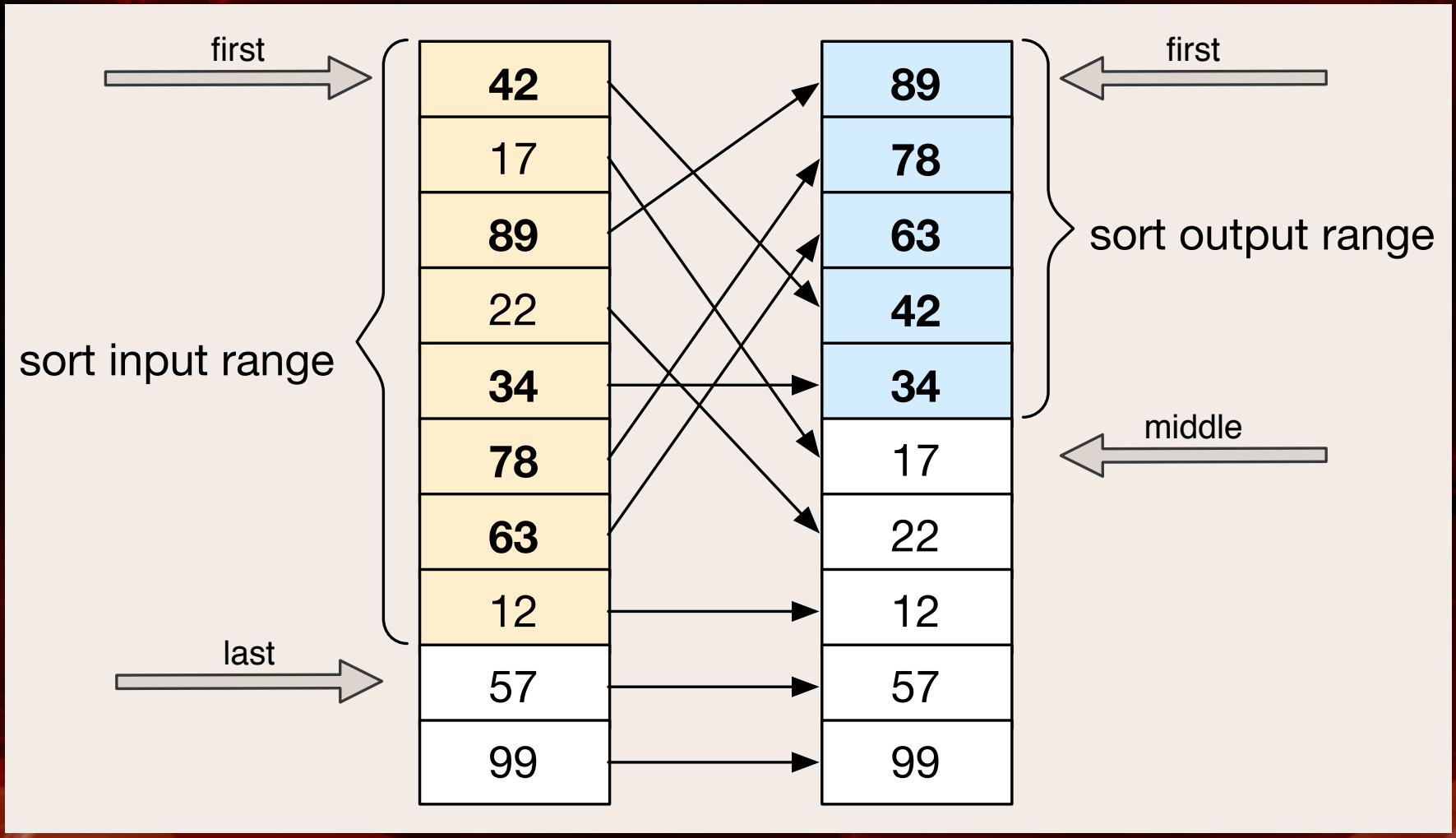
```
vector<int> v1{ 42, 17, 89, 22,
                34, 78, 63, 12,
                57, 99 };

partial_sort(v1.begin(),
             v1.begin() + 5,
             v1.begin() + 8,
             greater<int>());
```

partial_sort example

```
vector<int> v1{ 42, 17, 89, 22,
                34, 78, 63, 12,
                57, 99 };

partial_sort(v1.begin(),
             v1.begin() + 5,
             v1.begin() + 8,
             greater<int>());
```

partial_sort example

visual



Know your sorta-sorts

Scott Meyers' "Effective STL" Item 31

- `partition`
  - Reorder sequence so all items before partition point are less, and all items after are not less
  - Order of items in lower and upper subsets is implementation defined
  - Order of equivalent items is implementation-defined
  - Operates in place
- `stable_partition`
  - Like `partition`, but…
  - Order of equivalent items is preserved
  - Operates in place
- `partition_copy`
  - Like `partition`, but…
  - Order of equivalent items is implementation-defined
  - Copies items from input sequence to one of two output sequences depending on whether supplied function object returns false or true for each item
  - There is no `stable_partition_copy`

Know your sorta-sorts, cont.

Scott Meyers' "Effective STL" Item 31

- `nth_element`
  - Reorders sequence such that all items before "nth" are less, and all items after "nth" are not less
  - Order of items in lower and upper subsets is implementation defined
  - Order of equivalent items is implementation-defined
  - "nth" element is exactly the value that would exist in a fully sorted sequence (but without fully sorting)
  - Operates in place

Know your sorta-sorts, cont.

Comparison between partition and nth_element

`partition`

- partitions a sequence, based on condition
- partition point *is not* guaranteed to be value that would be at that position in fully sorted sequence
- input:
  - sequence begin, end
  - comparison function
- output:
  - reordered sequence
  - iterator to partition point

`nth_element`

- partitions a sequence, based on position
- nth element is element that *would* exist in that position in fully sorted sequence
- input:
  - sequence begin, end
  - iterator to “nth” position
  - optional comparison function
- output:
  - reordered sequence, partitioned around nth element

partition example

partition elements around 50

```
vector<int> v{ 12, 89, 31, 18,
               7, 72, 69, 50,
              49, 50, 51, 49 };

vector<int>::iterator part_it =
    partition(v.begin(),
              v.end(),
              [](const int i) {
                  return i < 50;
              });
```

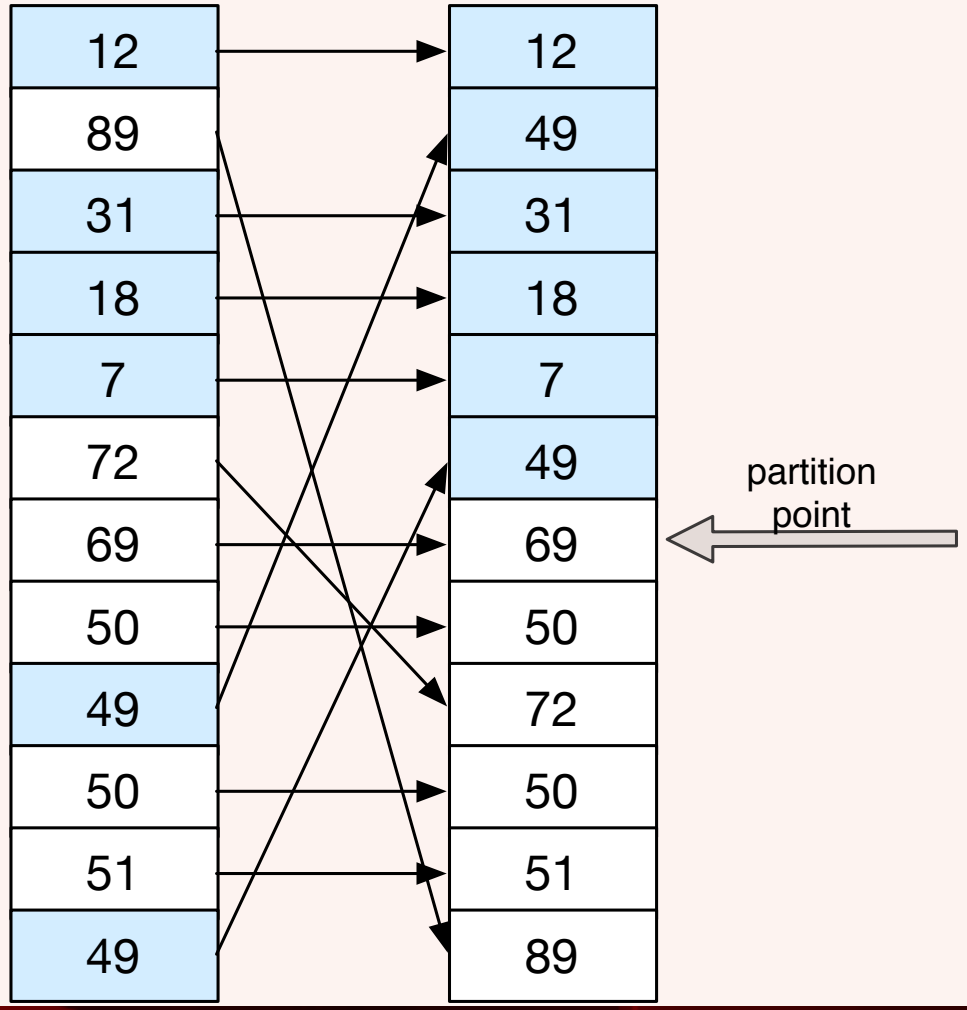
partition example

partition elements around 50

```
vector<int> v{ 12, 89, 31, 18,
               7, 72, 69, 50,
               49, 50, 51, 49 };

vector<int>::iterator part_it =
    partition(v.begin(),
              v.end(),
              [](const int i) {
                  return i < 50;
              });
```

partition example

visual: partition elements around 50



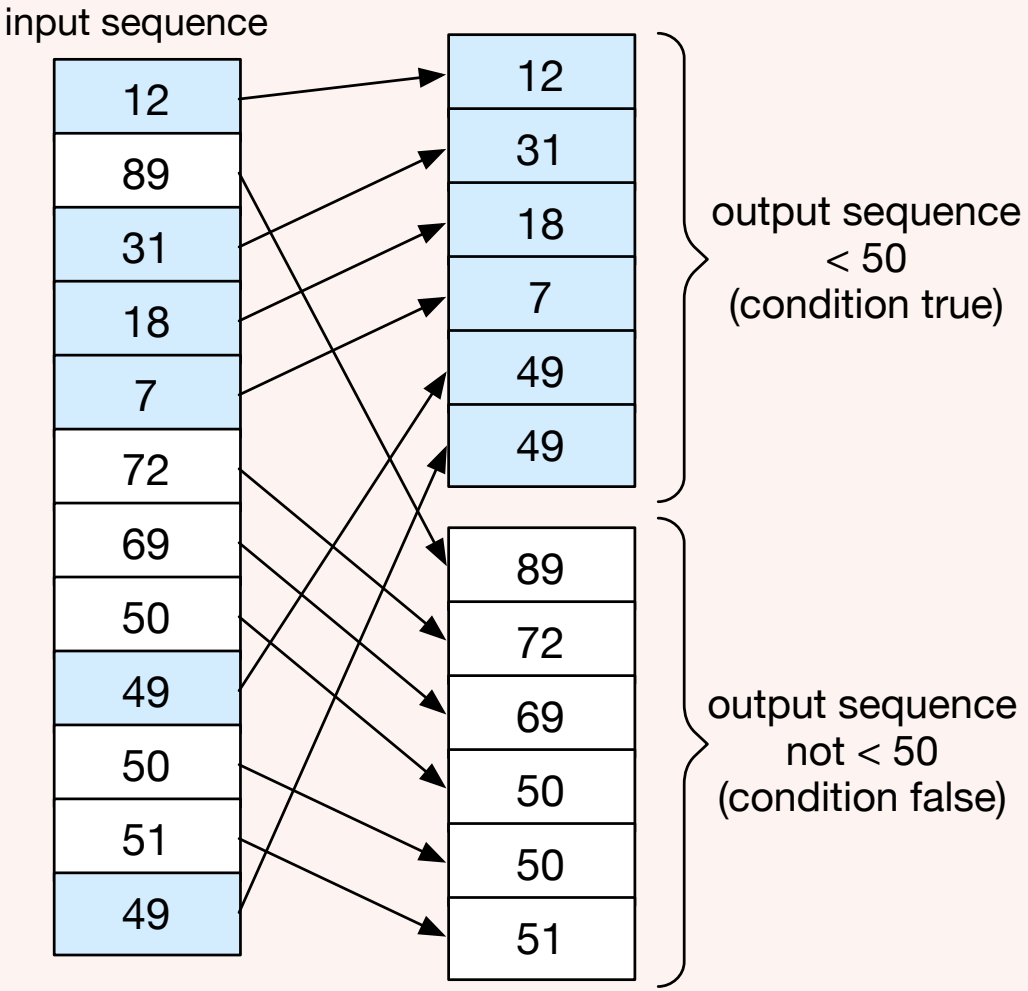
partition_copy example

partition elements around 50

```
vector<int> v{ 12, 89, 31, 18,
               7, 72, 69, 50,
              49, 50, 51, 49 };

partition_copy(v.begin(),
               v.end(),
               back_inserter(smaller),
               back_inserter(larger),
               [](const int i) {
                   return i < 50;
               });
```

`partition_copy` example

visual: partition elements around 50



Mixing up partitions!

`partition_copy`, `partition`, and `stable_partition` example

- Extend our prior Person struct to be an Employee struct
- Partition Management and Individuals into two separate containers
- Partition Management between Executives and Managers
- Partition Architects before other Individuals
- Stable partition the Senior and Junior employees among already partitioned individuals, leaving already partitioned elements in same relative position

Mixing up partitions!

some data...

```
struct Employee: public Person {
    enum class EmployeeType {
        Executive, Manager, Architect, Senior, Junior,
        Management = Manager,
        Individual = Architect
    };
    EmployeeType type;

    … additional members and methods…
};

vector<Employee> v{
   { "Joe", "P", "Smith", Employee::EmployeeType::Manager },
   { "Jane", "Q", "Jones", Employee::EmployeeType::Junior },
   { "Frank", "P", "Johnson", Employee::EmployeeType::Architect },
   { "Sarah", "B", "Smith", Employee::EmployeeType::Executive },
   { "Joe", "X", "Jones", Employee::EmployeeType::Senior },
   { "Joe", "A", "Smith", Employee::EmployeeType::Junior },
   { "Chris", "M", "Williams", Employee::EmployeeType::Manager }
};
```

Mixing up partitions!

separate employees into management and individuals

```
vector<Employee> management, individuals;
partition_copy(v.begin(), v.end(),
               back_inserter(management), back_inserter(individuals),
               [](const Employee& e) {
                   return e.type <= Employee::EmployeeType::Manager;
               });
```

executives get the company jet; managers get the company car

```
vector<Employee>::iterator car_it =
    partition(management.begin(), management.end(),
              [](const Employee& e) {
                  return e.type == Employee::EmployeeType::Executive;
              });
vector<Employee>::iterator jet_it = management.begin();

cout << "Management partitioned:\n";
for (auto it = management.begin(); it != management.end(); ++it) {
    cout << (it < car_it ? "\tjet" : "\tcar") << *it;
}
```

Mixing up partitions!

architects get the company segway; everyone else gets the bike

```
vector<Employee>::iterator bike_it =
    partition(individuals.begin(), individuals.end(),
              [](const Employee& e) {
                  return e.type == Employee::EmployeeType::Architect;
              });
vector<Employee>::iterator segway_it = individuals.begin();
cout << "Architects partitioned:\n";
for (auto it = individuals.begin(); it != individuals.end(); ++it) {
    cout << (it < bike_it ? "\tsegway" : "\tbike") << *it;
}
```

partition the non-architects from higher to lower seniority

```
vector<Employee>::iterator old_bike_it =
    stable_partition(individuals.begin(), individuals.end(),
                     [](const Employee& e) {
                         return e.type <= Employee::EmployeeType::Senior;
                     });
cout << "Individuals partitioned:\n";
for (auto it = individuals.begin(); it != individuals.end(); ++it) {
    cout << (it < bike_it ? "\tsegway" :
             (it < old_bike_it ? "\tnewbike" : "\toldbike")) << *it;
}
```

Mixing up partitions!

example output

```
Management partitioned:
      jet    Sarah B Smith: Executive
      car    Joe P Smith: Manager
      car    Chris M Williams: Manager
Architects partitioned (junior/senior unpartitioned):
      segway       Frank P Johnson: Architect
      bike         Jane Q Jones: Junior
      bike         Joe X Jones: Senior
      bike         Joe A Smith: Junior
Individuals fully partitioned:
      segway       Frank P Johnson: Architect
      newbike      Joe X Jones: Senior
      oldbike      Jane Q Jones: Junior
      oldbike      Joe A Smith: Junior
```

nth_element (not a super-hero, but still pretty super)

find median value in sequence

```
vector<int> v{ 12,  2, 89, 78, 18,  7, 72,
               69, 81, 50, 49, 50, 51, 49 };

const size_t nth = v.size() / 2;
nth_element(v.begin(), v.begin() + nth, v.end());
```

output:

```
49 2 12 49 18 7 50 >50< 51 78 72 69 81 89
```

find percentile value in sequence

```
vector<int> v{ 12,  2, 89, 78, 18,  7, 72,
               69, 81, 50, 49, 50, 51, 49 };

const size_t percentile = 75;
const size_t nth = v.size() * percentile / 100;
nth_element(v.begin(), v.begin() + nth, v.end());
```

output:

```
49 2 12 49 18 7 50 50 51 69 >72< 78 81 89
```

`partial_sort` **vs.** `nth_element` + `sort`

- Scenario: find top (say 20) web pages by number of hits, among a much larger set
- We have a large sequence where we are only concerned with the top 20 elements
- We want the top 20 elements of this sequence in order; we don't care about the rest
- Requirement: top 20 must be partitioned to front of sequence, before the don't-cares
- Requirement: top 20 must be sorted

partial_sort vs. nth_element + sort

```
partial_sort(vec.begin(),
             vec.begin() + 20,
             vec.end(),
             greater<int64_t>());


nth_element(vec.begin(),
            vec.begin() + 20,
            vec.end(),
            greater<int64_t>());
sort(vec.begin(), vec.begin() + 20,
     greater<int64_t>());
```

partial_sort vs. nth_element + sort

- Are they equivalent?

```
vector<int64_t> v1 = init_vec();
vector<int64_t> v2 = v1; // make a copy

partial_sort(v1.begin(), v1.begin() + 20, v1.end(), greater<int64_t>());

nth_element(v2.begin(), v2.begin() + 20, v2.end(), greater<int64_t>());
sort(v2.begin(), v2.begin() + 20, greater<int64_t>());

cout << "sorted portions of vectors are equal: " << boolalpha
     << equal(v1.begin(), v1.begin() + 20, v2.begin()) << endl;
cout << "unsorted portions of vectors are equal: " << boolalpha
     << equal(v1.begin() + 20, v1.end(), v2.begin() + 20) << endl;
```

output:

```
sorted portions of vectors are equal: true
unsorted portions of vectors are equal: false
```

rotate

- Moves elements specified in a source range to destination position, while moving displaced items to source position
- Simple way to move one or more elements from one position in a container to another without using erase and insert
- Technically, this is a left rotate
- `rotate` moves element in the sequence in-place
- `rotate_copy` does the same, except that it copies the elements to a separate sequence
- Requires:
  - `[first, middle)` and `[middle, last)` are valid ranges
  - i.e. `first <= middle <= last`

rotate

A naïve implementation – move one item in sequence

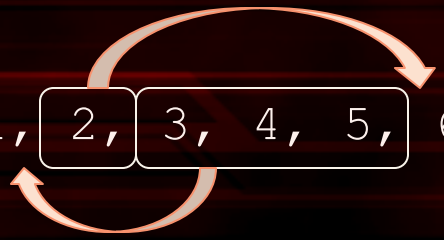
```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int>::iterator it = v.begin() + 2;
int i = *it;
v.erase(it);

vector<int>::iterator it2 = v.begin() + 6 - 1;
v.insert(it2, i);
```

Output:

```
before: 0 1 2 3 4 5 6 7 8 9
After:  0 1 3 4 5 2 6 7 8 9
```

rotate

A naïve implementation – move one item in sequence



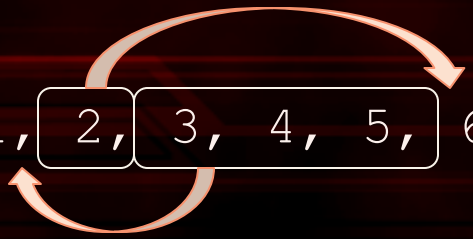
```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int>::iterator it = v.begin() + 2;
int i = *it;
v.erase(it);

vector<int>::iterator it2 = v.begin() + 6 - 1;
v.insert(it2, i);
```

Output:

```
before: 0 1 2 3 4 5 6 7 8 9
After:  0 1 3 4 5 2 6 7 8 9
```

rotate

A better implementation – move one item in sequence



```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int>::iterator it = v.begin() + 2;
vector<int>::iterator it2 = v.begin() + 6;
rotate(it, it + 1, it2);
```

Output:

```
before: 0 1 2 3 4 5 6 7 8 9
After:  0 1 3 4 5 2 6 7 8 9
```

rotate

Rotate a range of items

```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

vector<int>::iterator it = v.begin() + 2;
vector<int>::iterator it2 = v.begin() + 7;
rotate(it, it + 3, it2);
```

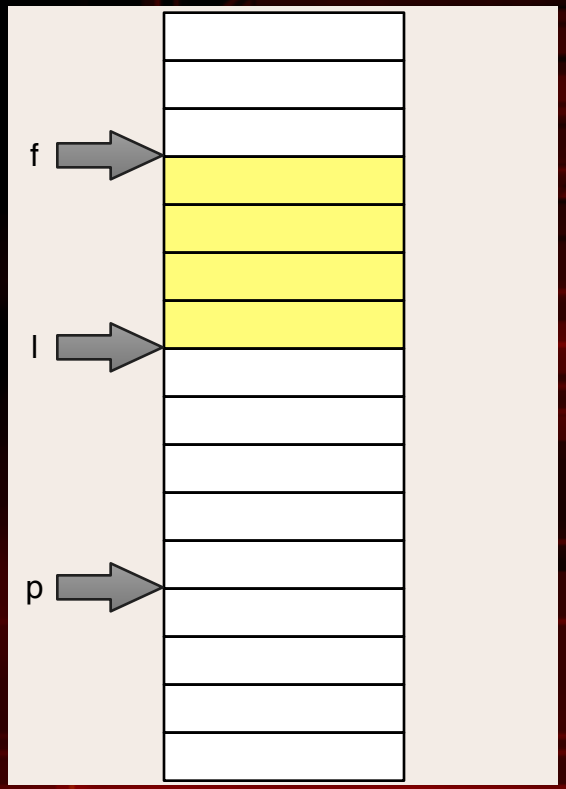
Output:

```
before:     0 1 2 3 4 5 6 7 8 9
after:      0 1 5 6 2 3 4 7 8 9
```

`rotate`
GCC's definition

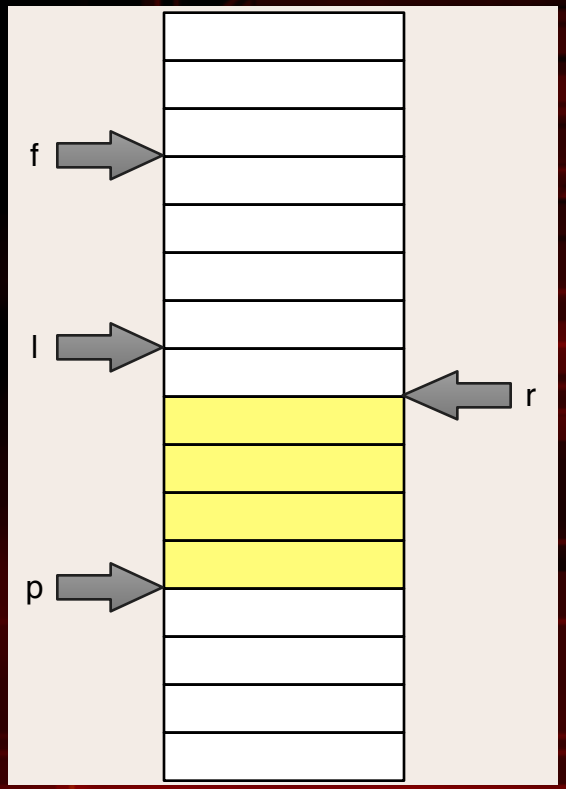
- Rotates the elements of the range `[first, last)` by `(middle - first)` positions so that the element at `middle` is moved to `first`, the element at `middle + 1` is moved to `first + 1` and so on for each element in the range.
- This effectively swaps the ranges `[first, middle)` and `[middle, last)`.

`slide` (move range of elements to new position)
Sean Parent, "C++ Seasoning"



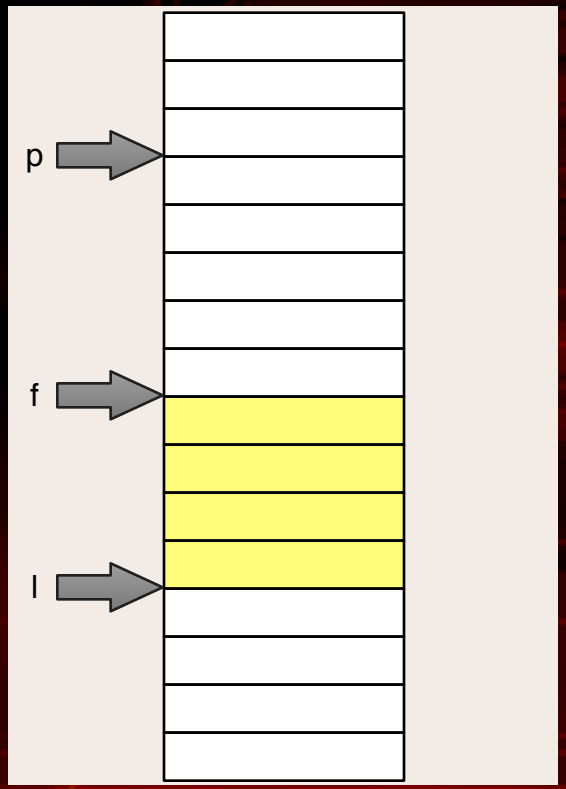
- What if you wanted to implement an algorithm that would move a range to a point in either direction without worrying about iterator ordering?
- For example, move a range of selected GUI items to a new position in a list.
- When using `rotate`, it is important to get the order of iterators correct: `first <= middle <= last`.
- Bonus: return a pair indicating destination range of relocated elements.

`slide` (move range of elements to new position)
Sean Parent, "C++ Seasoning"



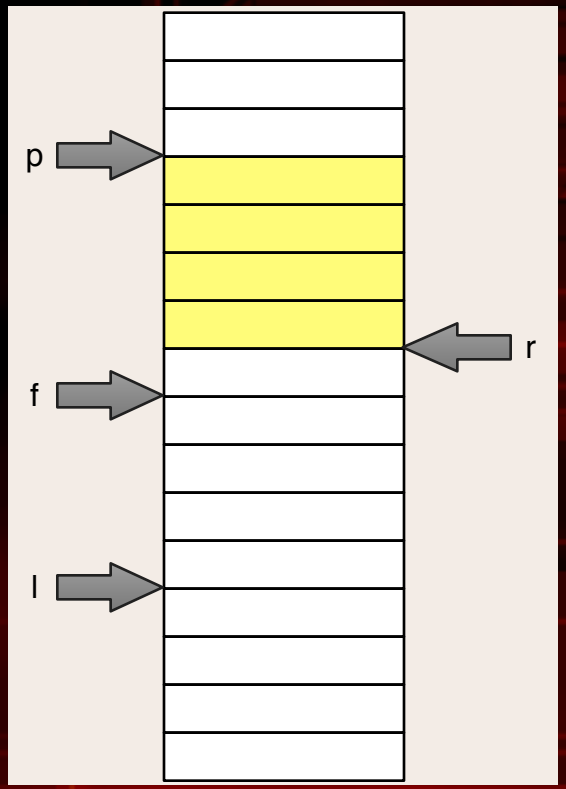
- What if you wanted to implement an algorithm that would move a range to a point in either direction without worrying about iterator ordering?
- For example, move a range of selected GUI items to a new position in a list.
- When using `rotate`, it is important to get the order of iterators correct: `first <= middle <= last`.
- Bonus: return a pair indicating destination range of relocated elements.

`slide` (move range of elements to new position)
Sean Parent, "C++ Seasoning"



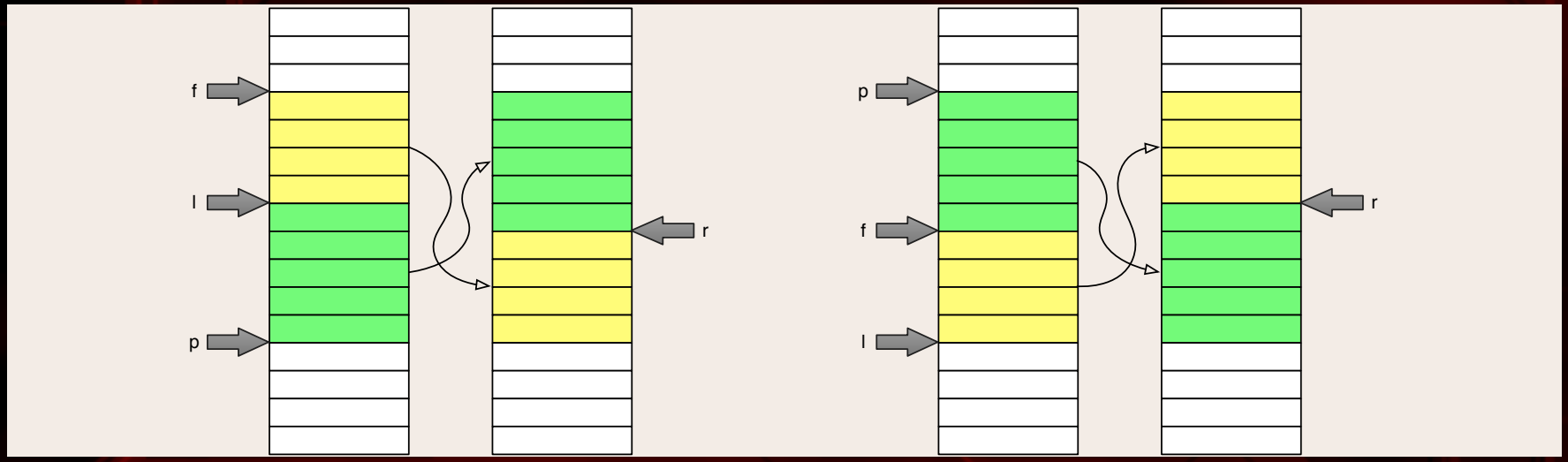
- What if you wanted to implement an algorithm that would move a range to a point in either direction without worrying about iterator ordering?
- For example, move a range of selected GUI items to a new position in a list.
- When using `rotate`, it is important to get the order of iterators correct: `first <= middle <= last`.
- Bonus: return a pair indicating destination range of relocated elements.

`slide` (move range of elements to new position)
Sean Parent, "C++ Seasoning"



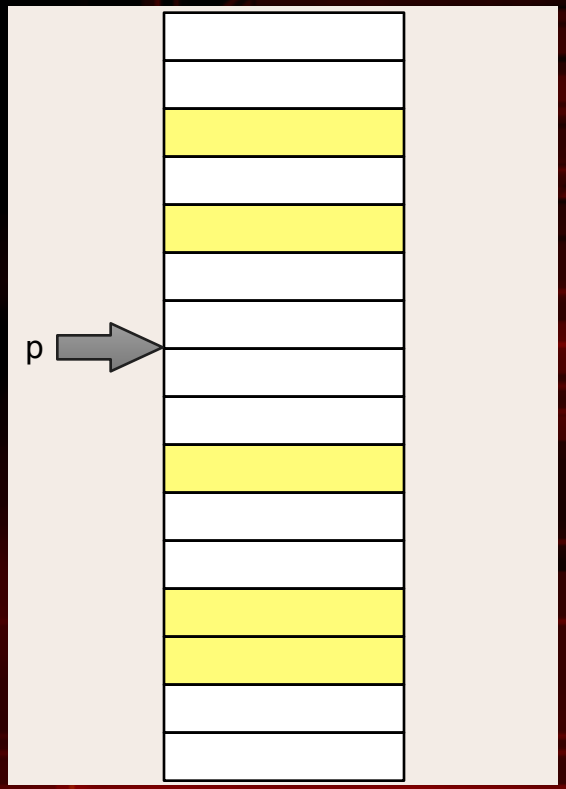
- What if you wanted to implement an algorithm that would move a range to a point in either direction without worrying about iterator ordering?
- For example, move a range of selected GUI items to a new position in a list.
- When using `rotate`, it is important to get the order of iterators correct: `first <= middle <= last`.
- Bonus: return a pair indicating destination range of relocated elements.

slide (move range of elements to new position)
Sean Parent, “C++ Seasoning”



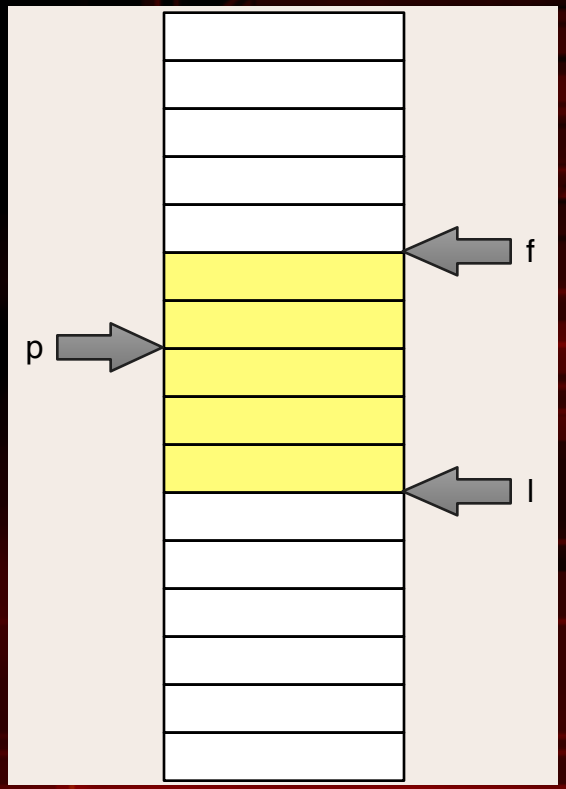

```
template <typename I> // I models RandomAccessIterator
auto slide(I f, I l, I p) -> pair<I, I> {
    if (p < f)
        return { p, rotate(p, f, l) };
    if (l < p)
        return { rotate(f, l, p), p };
    return { f, l };
}
```

gather (gather multiple elements around a point)
Sean Parent, "C++ Seasoning"



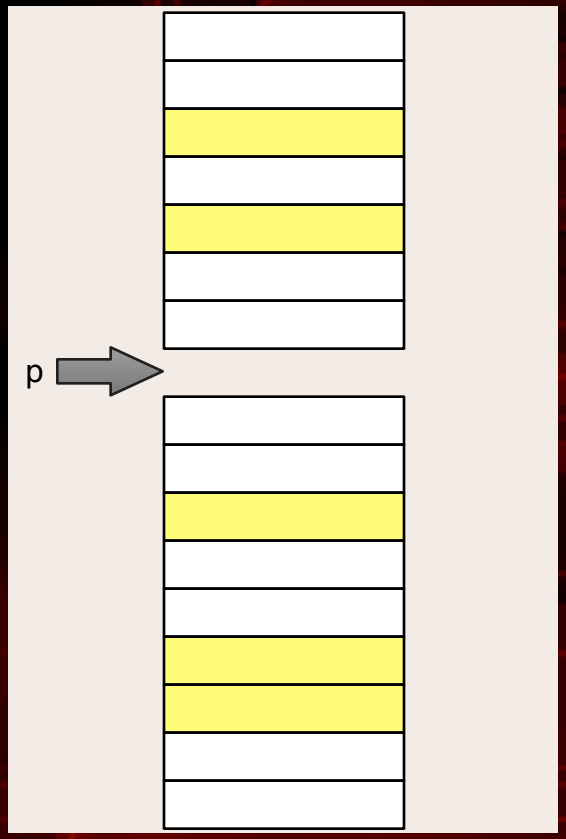
- What if you wanted to implement an algorithm that would allow a scattered selection of elements to be gathered to a specific location?
- For example, move a range of scattered multi-selected GUI items to a new position in a list, gathered together.
- How can you gather diverse elements to a single location without loops and special-casing code?
- Bonus: return a pair indicating destination range of relocated elements.

gather (gather multiple elements around a point)
Sean Parent, "C++ Seasoning"



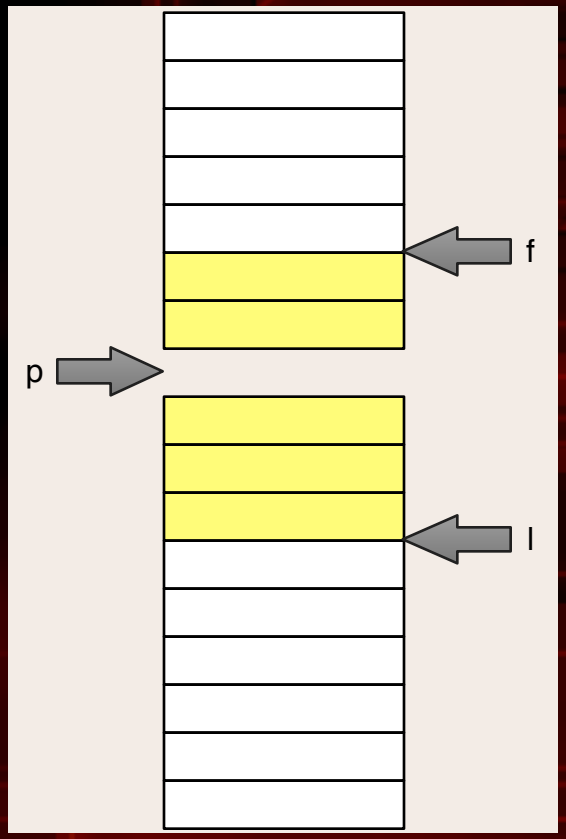
- What if you wanted to implement an algorithm that would allow a scattered selection of elements to be gathered to a specific location?
- For example, move a range of scattered multi-selected GUI items to a new position in a list, gathered together.
- How can you gather diverse elements to a single location without loops and special-casing code?
- Bonus: return a pair indicating destination range of relocated elements.

gather (gather multiple elements around a point)
Sean Parent, "C++ Seasoning"



- Break the problem space into two pieces:
  - The sequence before the destination position
  - The sequence after the destination position
- What algorithm can gather the select items while maintaining their relative position?

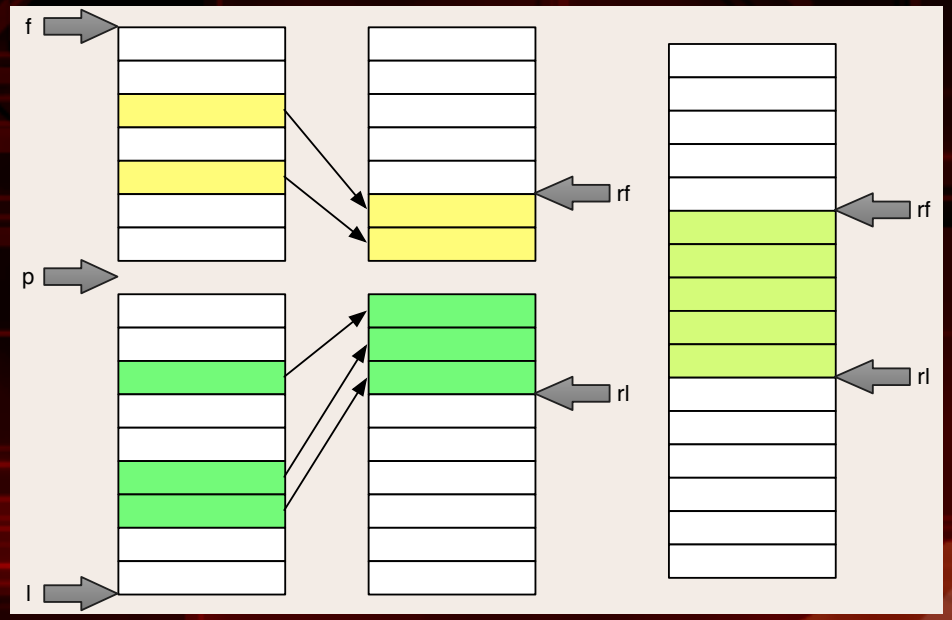
gather (gather multiple elements around a point)
Sean Parent, "C++ Seasoning"



- Break the problem space into two pieces:
  - The sequence before the destination position
  - The sequence after the destination position
- That looks a lot like partitioning…

gather (gather multiple elements around a point)
Sean Parent, “C++ Seasoning”

```
template <typename I, // models BidirectionalIterator
          typename S> // models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
```



set_difference

process configuration updates

- Scenario: you store some configuration information in an ordered container
- You receive updated configuration on a regular basis
- You want to quickly determine what has changed: i.e. what new items have been added, and what items have been removed

set_difference

process configuration updates

```
vector<string> new_items, removed_items;

set_difference(current.begin(), current.end(),
               update.begin(), update.end(),
               back_inserter(removed_items));
remove_config(removed_items);

set_difference(update.begin(), update.end(),
               current.begin(), current.end(),
               back_inserter(new_items));
add_config(new_items);
```

set_difference

process configuration updates test example

```
set<string> current { "one", "two", "three", "four", "five" };
set<string> update { "one", "three", "four", "six", "seven" };

vector<string> new_items, removed_items;

set_difference(current.begin(), current.end(),
               update.begin(), update.end(),
               back_inserter(removed_items));
remove_config(removed_items);

set_difference(update.begin(), update.end(),
               current.begin(), current.end(),
               back_inserter(new_items));
add_config(new_items);
```

output:

```
Removed: five two
Added: seven six
```

Coming soon to a compiler near you!
Parallel algorithms

- Currently experimental, proposed for a future release.
- In the `std::experimental::parallel` namespace.
- Have the same interfaces as standard algorithms.
- Controlled by use of parallelism `execution_policy`.
- See the Parallelism Technical Specification for more info.

A quick review…

- Be familiar with all the varied algorithms provided for free with your compiler
- Write some code that exercises each of them so you are familiar with their usage and individual personalities
- Write your own adaptations of existing algorithms
- Implement your own algorithms

STL algorithms

non-modifying sequence operations

- all_of
- any_of
- none_of
- for_each
- find
  - find_if
  - find_if_not
- find_end
- find_first_of
- adjacent_find
- count
  - count_if
- mismatch
- equal
- is_permutation
- search
  - search_n

STL algorithms
mutating sequence operations

- copy
  - copy_n
  - copy_if
  - copy_backward
- move
  - move_backward
- swap_ranges
  - iter_swap
- transform
- replace
  - replace_if
  - replace_copy
  - replace_copy_if
- fill
  - fill_n
- generate
  - generate_n
- remove
  - remove_if
  - remove_copy
  - remove_copy_if
- unique
  - unique_copy
- reverse
  - reverse_copy
- rotate
  - rotate_copy
- shuffle
  - random_shuffle
- partition
  - is_partitioned
  - stable_partition
  - partition_copy
  - partition_point

STL algorithms
sorting and related operations

- sorting
  - sort
  - stable_sort
  - partial_sort
  - partial_sort_copy
- nth_element
- binary search
  - lower_bound
  - upper_bound
  - equal_range
  - binary_search
- merge
  - merge
  - inplace_merge
- set operations on sorted structures
  - includes
  - set_union
  - set_intersection
  - set_difference
  - set_symmetric_difference
- heap operations
  - push_heap
  - pop_heap
  - make_heap
  - sort_heap
- minimum and maximum
  - min
  - max
  - minmax
  - min_element
  - max_element
  - minmax_element
- lexicographical comparisons
  - lexicographical_compare
- permutation generators
  - next_permutation
  - prev_permutation

STL algorithms
general numeric operations

- accumulate
- inner_product
- partial_sum
- adjacent_difference
- iota

Shout out to my “sponsor”

- F5 Networks is a highly technical company with a belief in well engineered software
- F5 Networks has graciously sent me here, and they ~~tolerate~~ encourage me working on this stuff in addition to my “real” work
- If you’re looking for something cool and challenging to do with C++, check out F5!
- https://f5.com/about-us/careers

Shameless advertising…

- I'm working on a book on this very topic!
- It isn't done yet…
- If you have some creative or clever uses of STL algorithms you'd like me consider, please drop me a line with some example code!
- michaelv@codeache.net
- Visit my blog for more adventures: http://codeache.net