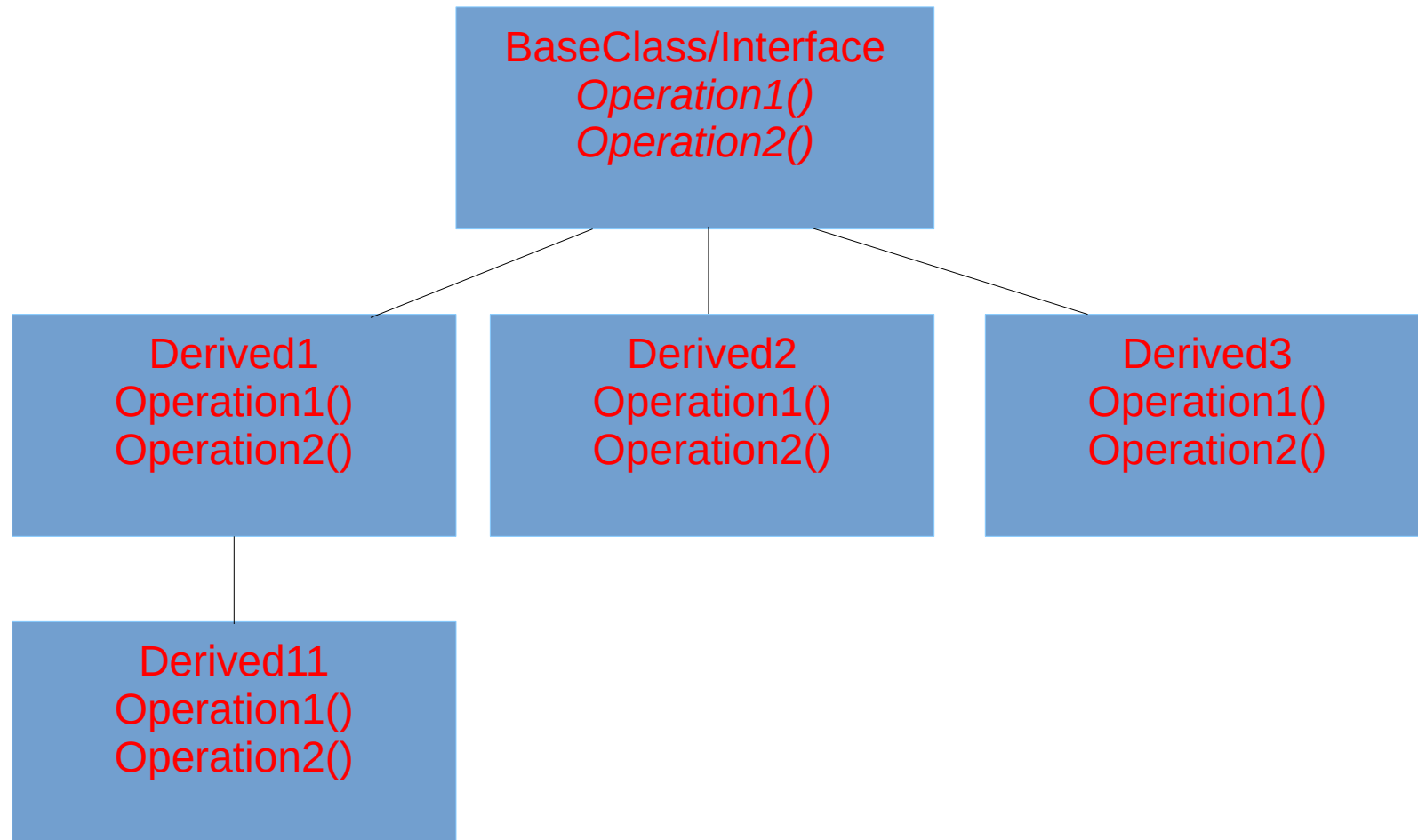


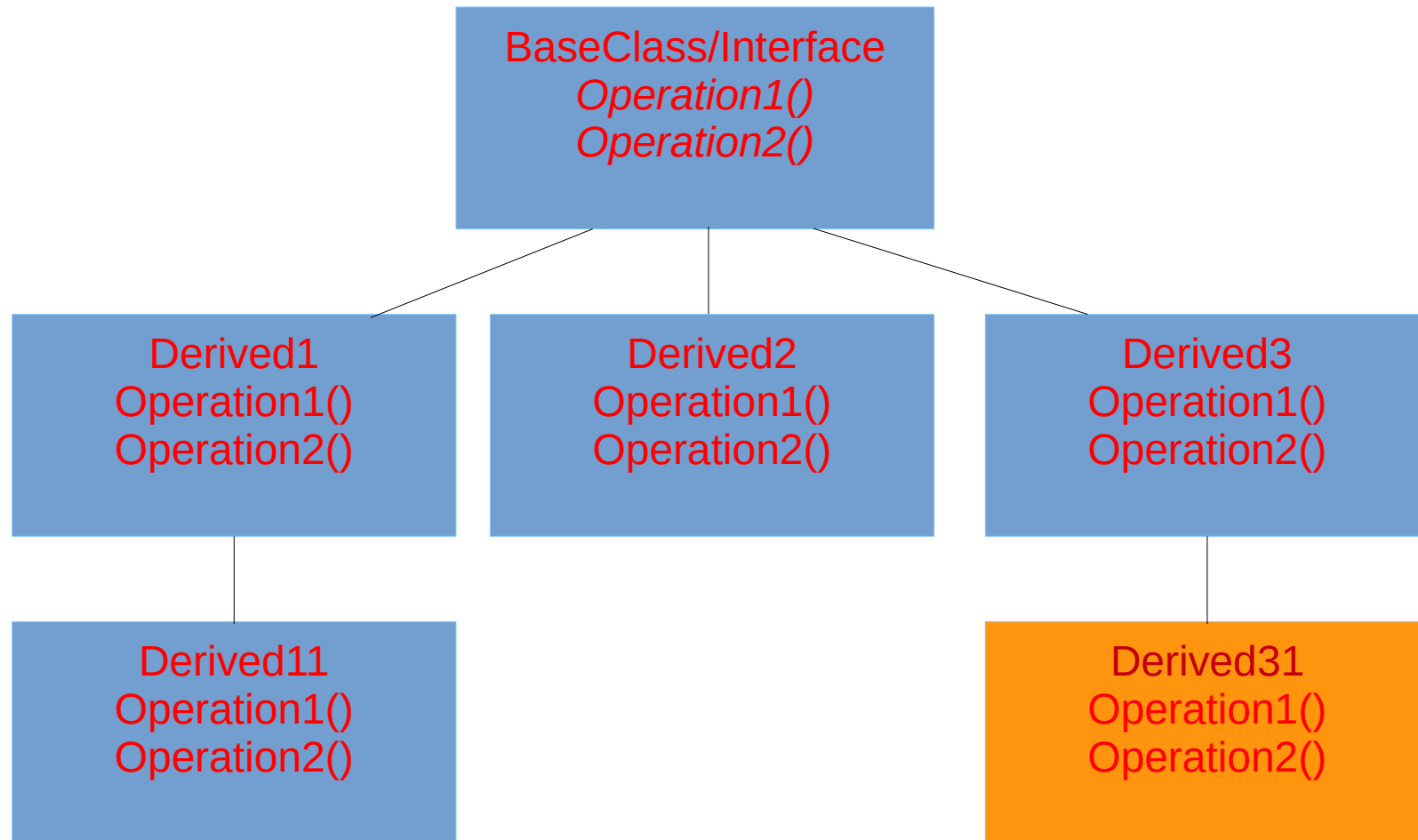
# STL

- Expression problem
- Generic programming
- An example – inserters, iterator-adapters, functors
- Efficiency
- Memory consumption characteristics
- Array and `forward_list` in C++11
- Unordered containers in C++11
- Traps and pitfalls

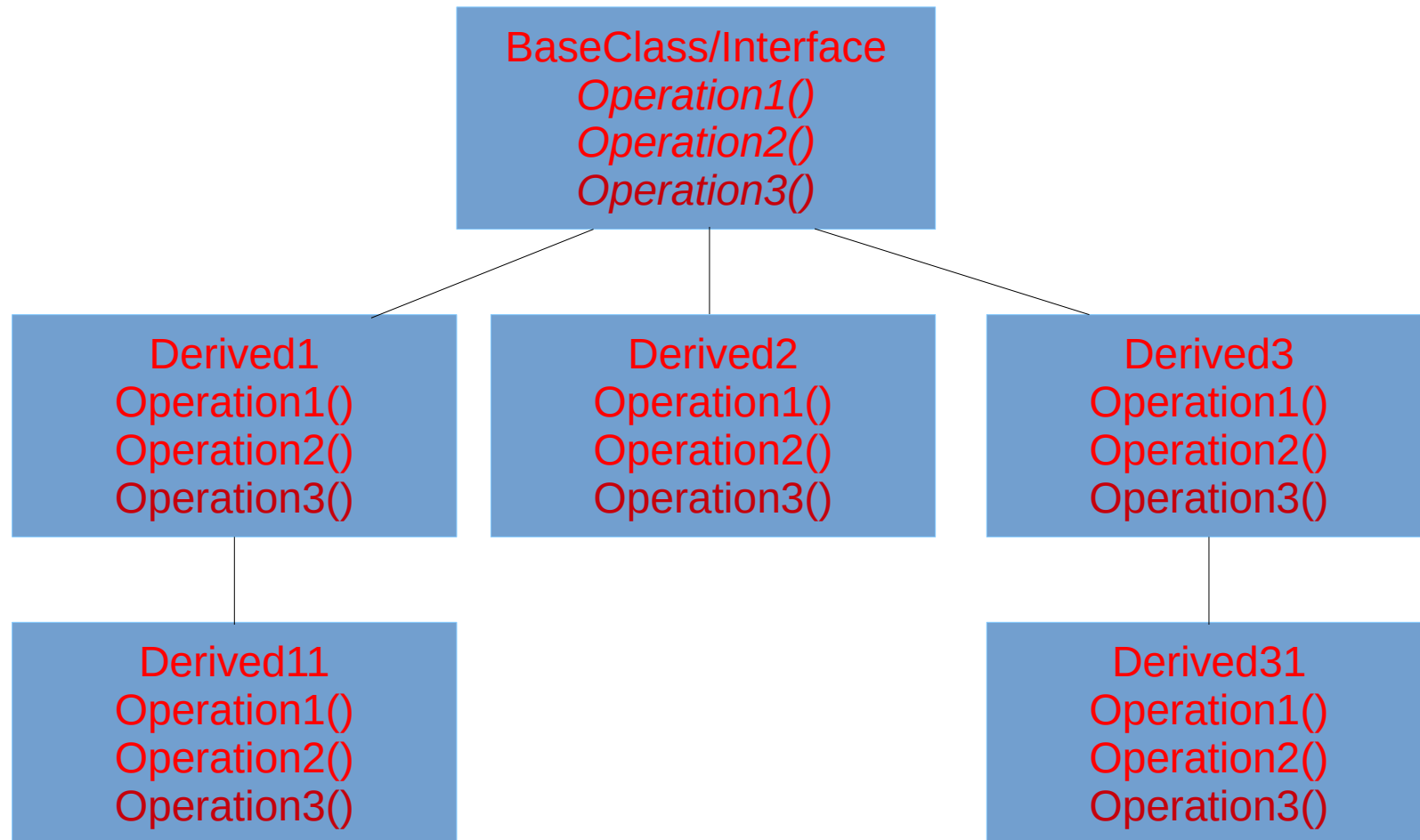
# The expression problem



# The expression problem



# The expression problem

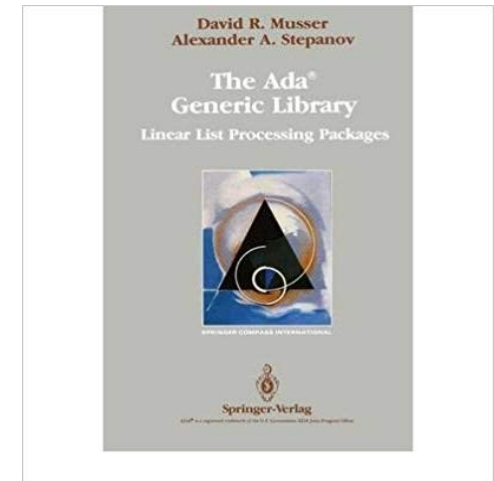
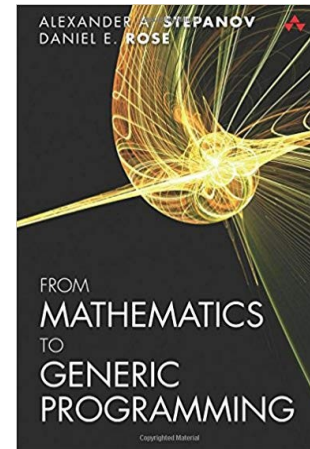
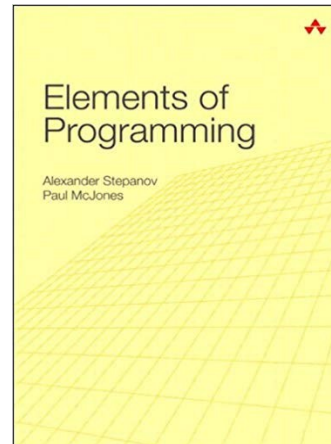


# The expression problem

- Philip Wadler: expression problem mail, 1990
- Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman: "Synthesizing Object-Oriented and Functional Design to Promote Re-Use", 1998

# The expression problem

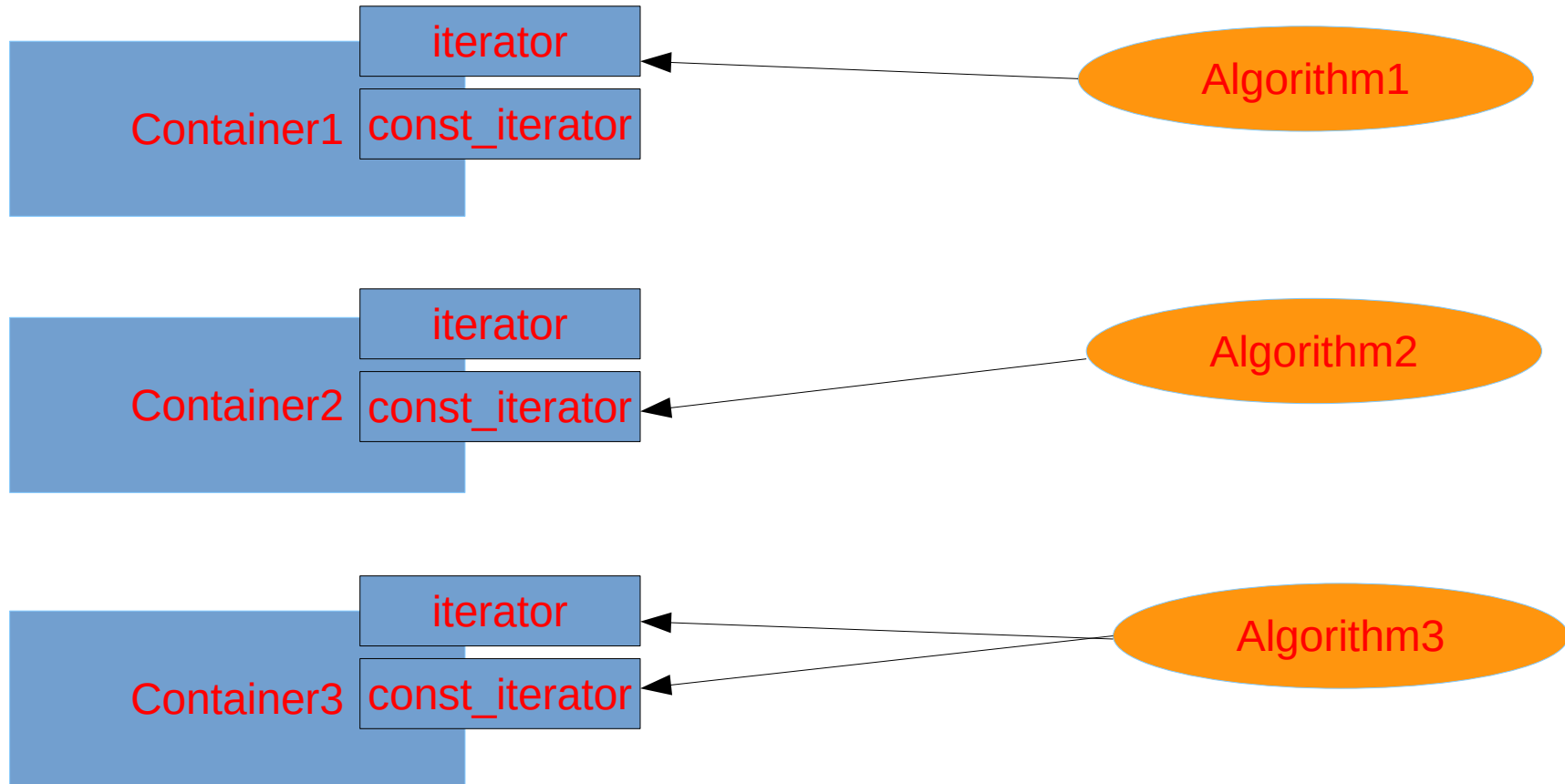
- Philip Wadler: expression problem mail, 1990
- Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman: "Synthesizing Object-Oriented and Functional Design to Promote Re-Use", 1998
- Aleksey Stepanov: generic programming, 1985



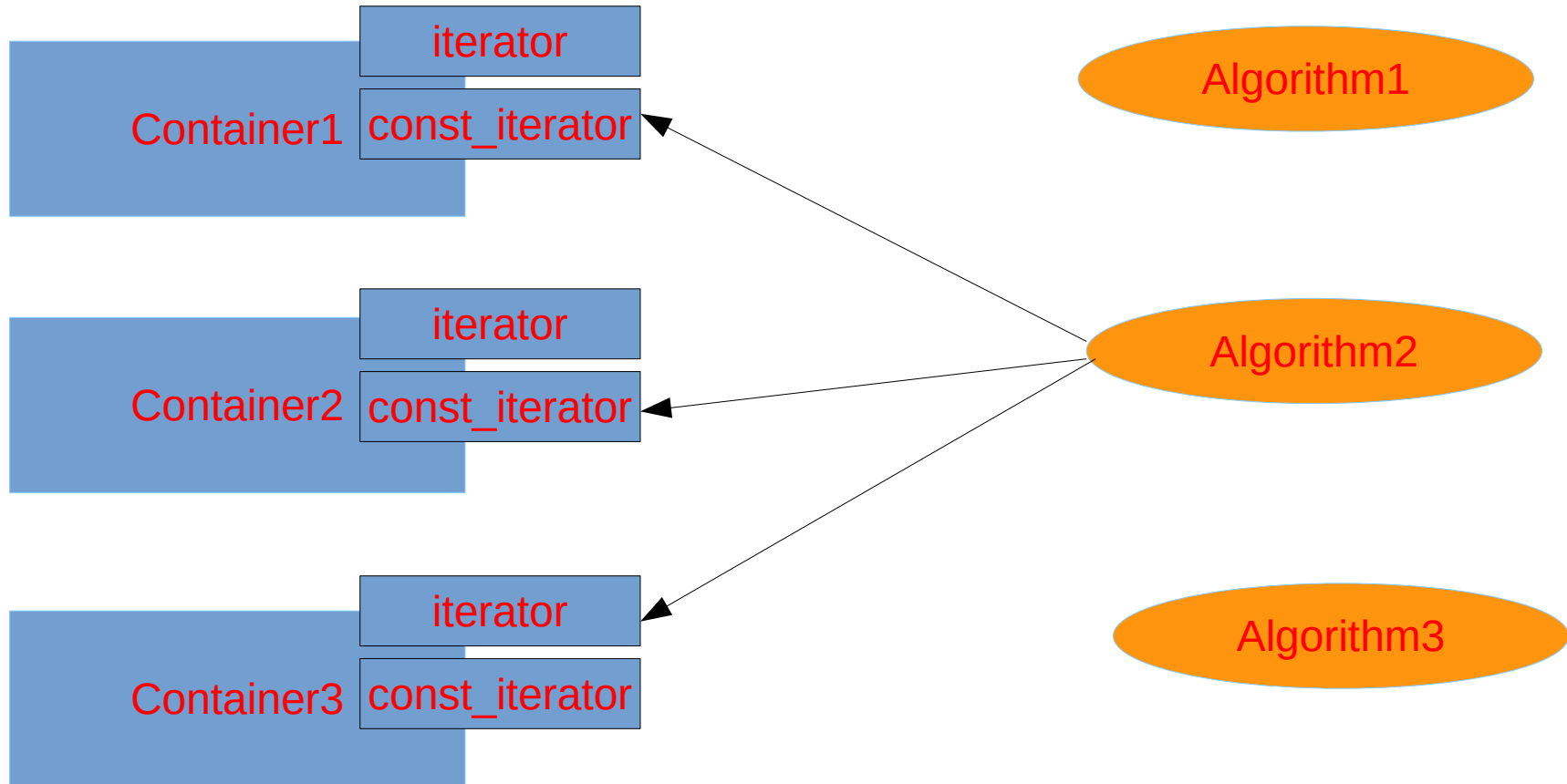
# The STL solution



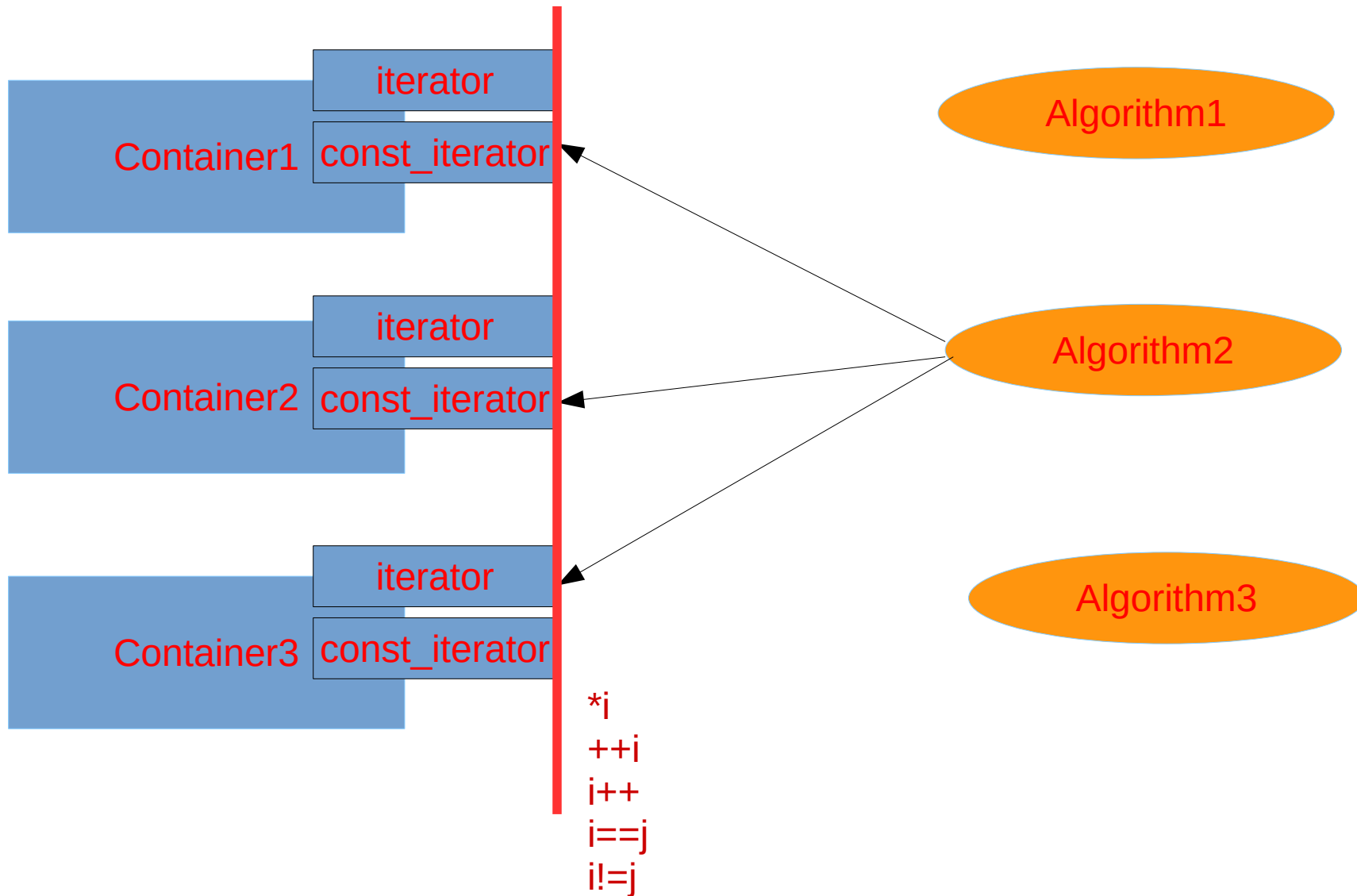
# The STL solution



# The STL solution



# The STL solution



# How to implement

```
int t[] = { 1, 3, 5, ... };

// find the first occurrence of value 55
int *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55);

if ( pi )
{
    *pi = 56
}

// a very specific algorithm: works only on integer arrays
int *find( int *begin, int *end, int x)
{
    while ( begin != end )
    {
        if ( *begin == x )
        {
            return begin;
        }
        ++begin;
    }
    return nullptr;
}
```

# Simple solution

```
int t[] = { 1, 3, 5, ... };  
  
// find the first occurrence of value 55  
int *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55);  
  
if ( pi )  
{  
    *pi = 56  
}
```

# Template based

```
double t[] = { 1.0, 3.14, 5.55, ... };

// find the first occurrence of a value
double *pi = find( t, t+sizeof(t)/sizeof(t[0]), 55.5);

if ( pi )
{
    *pi = 56.5
}

// Templated algorithm
template <typename T>
T *find( T *begin, T *end, const T& x)
{
    while ( begin != end )
    {
        if ( *begin == x )
        {
            return begin;
        }
        ++begin;
    }
    return nullptr;
}
```

# Iterator based

```
std::list<int> li = { 1, 3, 5, ... };

// find the first occurrence of value 55
auto it = find( li.begin(), li.end(), 55);

if ( li.end() != it )
{
    *it = 56
}

// Iterator based algorithm
template <typename It, typename T>
It find( It begin, It end, const T& x)
{
    while ( begin != end )
    {
        if ( *begin == x )
        {
            return begin;
        }
        ++begin;
    }
    return end; // not nullptr
}
```

# Universal usage

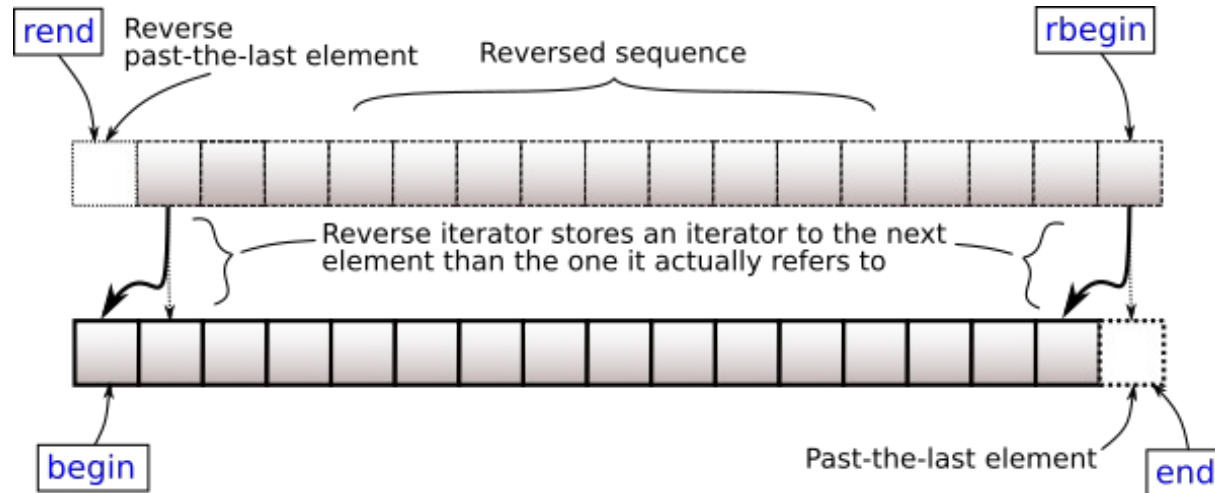
```
std::list<int> li = { 1, 3, 5, ... };  
std::vector<double> vd = { 1.0, 3.3, 5.5, ... };
```

```
template <typename Container>  
auto generic_find( const Container& c, typename Container::value_type& v )  
{  
    return std::find( c.begin(), c.end(), v );  
}
```

- ```
template<class C>  
typename C::value_type generic_sum(const C& c)  
{  
    typename C::value_type s{}; // value construct  
    typename C::const_iterator it = c.begin(); // auto it = c.begin()  
  
    while ( it != c.end() )  
    {  
        s += *it; // requires operator+= on C::value_type  
        ++it;  
    }  
    return s;  
}
```

```
auto i = generic_find(li, 5);  
auto d = generic_sum(vd);
```

# Reverse iterator



```
template<class C>
typename C::iterator find_last(C& c, const typename C::value_type& v)
{
    typename C::reverse_iterator p = c.rbegin(); // view sequence in reverse
    while ( p != c.rend() )
    {
        if ( *p == v )
        {
            typename C::iterator i = p.base();
            return --i;
        }
        ++p; // note: increment, not decrement (--)
    }
    return c.end(); // use c.end() to indicate "not found"
}
```

# Iterators are const safe

```
// C++11
std::vector<int> v1 = { 1, 2, 3, 4, ... };
auto i = std::find( v1.begin(), v1.end(), 3);
// i is vector::iterator

const std::vector<int> v2 = { 1, 2, 3, 4, ... };
auto j = std::find( v2.begin(), v2.end(), 3);
// j is vector::const_iterator

auto k = std::find( v1.cbegin(), v1.cend(), 3);
// k is vector::const_iterator

auto j2 = std::find( std::begin(v1), std::end(v1), 3);
// j2 is vector::iterator

auto k2 = std::find( std::cbegin(v1), std::cend(v1), 3);
// k2 is vector::const_iterator
```

# STL is const safe

```
template <typename It, typename T>
It find( It begin, It end, const T& t)
{
    while (begin != end) {
        if ( *begin == t )
            return begin;
        ++begin;
    }
    return end;
}

const char t[] = { 1, 2, 3, 4, 5 };
const char *p = std::find( t, t+sizeof(t), 3)
if ( p != t+sizeof(t) )
{
    std::cout << *p; // ok to read
    // syntax error: *p = 6;
}
const std::vector<int> v(t, t+sizeof(t));
std::vector<int>::const_iterator i = std::find( v.begin(), v.end(), 3);
if ( v.end() != i )
{
    std::cout << *i; // ok to read
    // syntax error: *i = 6;
}
```

# Functors

```
std::list<int> li = { 1, 3, 5, ... };

// find the third occurrence of value less than 55
auto it = find_if( li.begin(), li.end(), ? );

if ( li.end() != it )
{
    *it = 56
}

// Iterator based algorithm
template <typename It, typename Pred>
It find_if( It begin, It end, Pred p )
{
    while ( begin != end )
    {
        if ( p(*begin) )
        {
            return begin;
        }
        ++begin;
    }
    return end;
}
```

# Functors

```
std::list<int> li = { 1, 3, 5, ... };

// find the third occurrence of value less than 55
bool less55_3rd(int x)
{
    static int cnt = 0;
    if ( x < 55 ) ++cnt;
    return 3 == cnt;
}

// Iterator based algorithm
template <typename It, typename Pred>
It find_if( It begin, It end, Pred p)
{
    while ( begin != end )
    {
        if ( p(*begin) ) // calls less55_3rd(*begin)
        {
            return begin;
        }
        ++begin;
    }
    return end;
}
```

# Functors

```
std::list<int> li = { 1, 3, 5, ... };
```

```
bool less55_3rd(int x)
{
    static int cnt = 0;
    if ( x < 55 ) ++cnt;
    return 3 == cnt;
}
```

```
// find the third occurrence of value less than 55
auto it = find_if( li.begin(), li.end(), less55_3rd);
```

```
if ( li.end() != it )
{
    *it = 56;
}
```

# Functors

```
std::list<int> li = { 1, 3, 5, ... };
```

```
bool less55_3rd(int x)
{
    static int cnt = 0;
    if ( x < 55 ) ++cnt;
    return 3 == cnt;
}
```

```
// find the third occurrence of value less than 55
auto it = find_if( li.begin(), li.end(), less55_3rd);
```

```
if ( li.end() != it )
{
    *it = 56;
    it = find_if( ++it, li.end(), less55_3rd); // works?
}
```

# Functors

```
struct less55_3rd
{
    less55_3rd() : cnt(0) { }
    bool operator()(int x)
    {
        if ( x < 55 )    ++cnt;
        return 3 == cnt;
    }
private:
    int cnt;
};
```

```
template <typename It, typename Pred>
It find_if( It begin, It end, Pred p)
{
    while ( begin != end )
    {
        if ( p(*begin) ) // calls p.operator()( *begin)
        {
            return begin;
        }
        ++begin;
    }
    return end;
}
```

# Functors

```
struct less55_3rd
{
    less55_3rd() : cnt(0) { }
    bool operator()(int x)
    {
        if ( x < 55 )    ++cnt;
        return 3 == cnt;
    }
private:
    int cnt;
};
```

```
// find the third occurrence of value less than 55
auto it = find_if( li.begin(), li.end(), less55_3rd{});
```

```
if ( li.end() != it )
{
    *it = 56;
    it = find_if( ++it, li.end(), less55_3rd{}); // new object, cnt = 0
}
```

# Functors

```
struct less55_3rd
{
    less55_3rd() : cnt(0) { }
    bool operator()(int x)
    {
        if ( x < 55 )    ++cnt;
        return 3 == cnt;
    }
private:
    int cnt;
};
```

```
// find the third occurrence of value less than 55
```

```
auto it = find_if( li.begin(), li.end(), [](int x) { static int cnt = 0;
  if(x < 55) ++cnt;
  return 3 == cnt;} );

if ( li.end() != it )
{
    *it = 56;
    it = find_if( ++it, li.end(), [](int x) { ... }); // new lambda, cnt = 0
}
```

# Functors

```
struct less55_3rd
{
    less55_3rd() : cnt(0) { }
    bool operator()(int x)
    {
        if ( x < 55 )    ++cnt;
        return 3 == cnt;
    }
private:
    int cnt;
};

// find the third occurrence of value less than 55
auto it = find_if( li.begin(), li.end(), [cnt=0](int x) { // since C++14
   if(x < 55) ++cnt;
   return 3 == cnt;} );

if ( li.end() != it )
{
    *it = 56;
    it = find_if( ++it, li.end(), [cnt=0](int x) { ... }); // new lambda
}
```

# Functors

```
template <typename T>
struct less_Nth
{
    less_Nth(const T& t, int n) : value(t), nth(n), cnt(0) { }
    bool operator()(const T& x)
    {
        if ( x < value )    ++cnt;
        return nth == cnt;
    }
private:
    T    value;
    int  nth;
    int  cnt;
};

// find the fifth occurrence less than value 3.14
auto it = find_if( li.begin(), li.end(), less_Nth<double>{3.14,5});

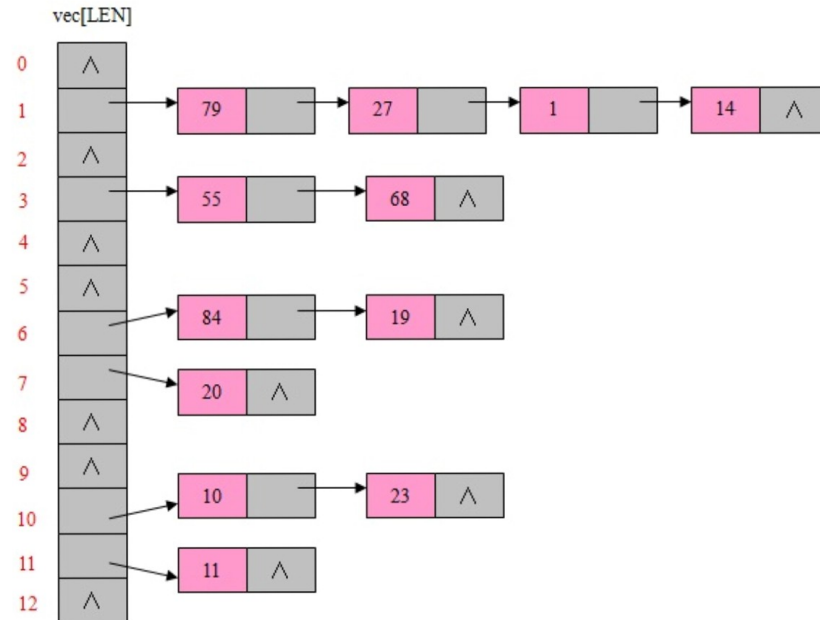
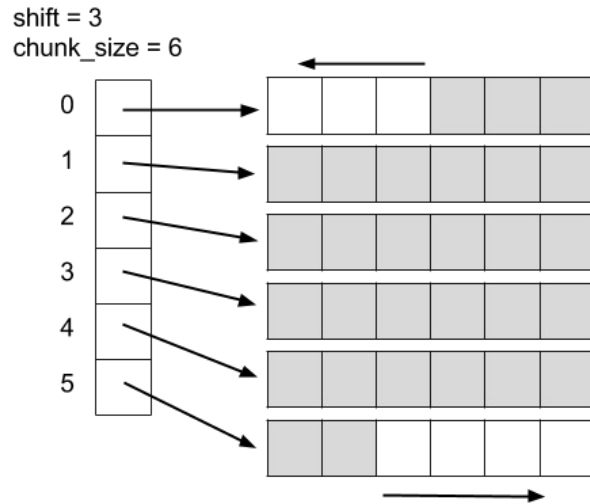
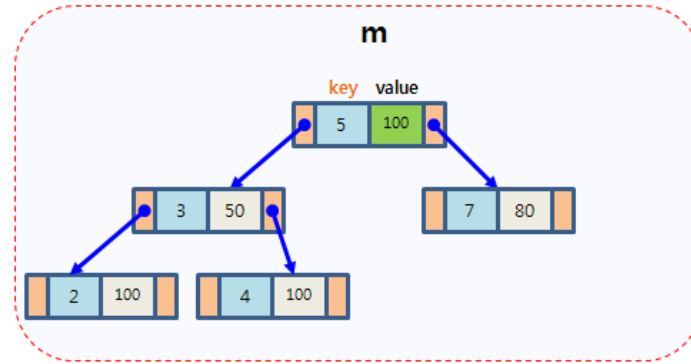
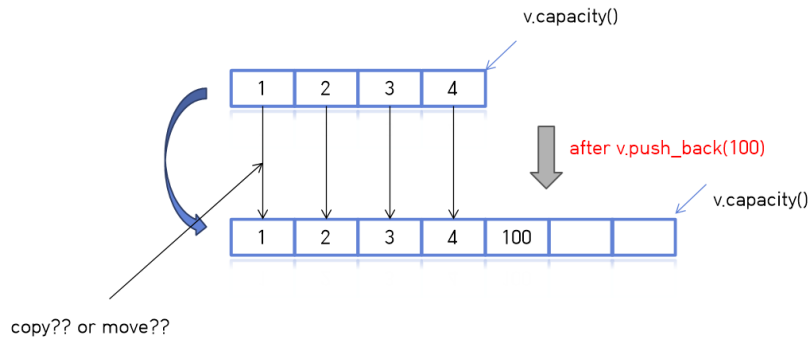
if ( li.end() != it )
{
    *it = 2.178;
    it = find_if( ++it, li.end(), less_Nth<double>{99.9,7});
}
```

# Standard containers

- Sequential containers
  - array, vector, deque, forward\_list, list, basic\_string\*
- Associative containers
  - set, map, multiset, mutimap
- Unordered associative (hash) containers
  - unordered\_map, ...\_set, ...\_multimap, ...\_multiset
- Container adaptors
  - stack, queue, priority\_queue
- Views
  - span, basic\_string\_view\*

\*officially not container, but very much behaves like it

# Typical container implementations



# Std::array

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // construction uses aggregate initialization
    std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 (not in C++14)
    std::array<int, 3> a2 = {1, 2, 3}; // never required after =
    std::array<std::string, 2> a3 = { std::string("a"), "b" };

    // container operations are supported
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

    std::cout << '\n';

    // ranged for loop is supported
    for(const auto& s: a3)
        std::cout << s << ' ';

    std::array a1{"foo"}; // C++17 CTAD: std::array<const char*,1>{"foo"}
    auto a2 = std::to_array{"foo"}; // std::array<char,4>{'f','o','o','\0'};
}
```

# Std::vector

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <vector>

int main()
{
    std::vector<int> v0; // 0 element vector of int
    std::vector<int> v5 = {1,2,3,4,5}; // 5 element vector of int
    std::vector<int> v9(9,42); // 9 element vector of int 42

    assert( v0.empty() && v0.size() == 0 && std::size(v0) && std::ssize(v0) == 0 )

    v0.push_back(1); // cheap O(1) insertion to the back
    v0.emplace_back(2); // cheap O(1) creation at the back

    assert( !v0.empty() && v0.size() == 2 && std::size(v0) && std::ssize(v0) == 2 );

    v0::value_type i1 = v0.front(); // 1 first element
    std::vector<int>::value_type i2 = v0.back(); // 2 last element

    v0.pop_back(); // remove last item 2
    v0.pop_back(); // remove last item 1
    v0.pop_back(); // undefined behavior, v0.empty() == true

    return v2 == v1; // lexicographical comparision, similarly !=, <, <=, ...
}
```

# Std::vector

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <vector>

void f(std::vector<int> v)
{
    v.at(3) = 42;    // throws std::out_of_range if v.size() < 4
    v[3] = 43;      // does not throw, undefined behavior if v.size() < 4

    std::vector<int> v2{v1} // copy elements, v2 == v1

    for ( auto a : v2 ) { std::cout << a << ", " }
    for ( auto it = v2.begin(); it != v2.end(); ++it) { std::cout << *it << ", " }
    for ( auto it = std::begin(v2); it != std::end(v2); ++it) { std::cout << *it << ", " }

    v2::iterator i = std::begin(v2)+4;
    v2.insert( i, 42); // insert into the middle, shift elements back, may resize
    *i = 43; // oops, iterator may be invalidated

    v2.erase( std::begin(v2), std::begin(v2)+2); // erase v2[0], v2[1] from v2
    i = std::find( std::begin(v2), std::end(v2), 42); // find first occurrence of 42
    if ( i != v2.end() ) // found
        *i = 43;
    v2.push_back(44); // may resize
    *i = 45; // oops, iterator may be invalidated
}
```

# Std::vector

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <vector>

int f()
{
    std::vector<int> v;
    std::cout << v.size() << '\n'    // 0

    v.reserve(1000);    // pre-allocate buffer with >= 1000 capacity
    std::cout << v.capacity() << '\n'    // >= 1000

    v.resize(1000);    // add 1000 default int: 0
    std::cout << v.size() << '\n'    // 1000

    v.resize(10);    // erase the last 990 elements
    std::cout << v.size() << '\n'    // 10
    std::cout << v.capacity() << '\n'    // >= 1000

    v.shrink_to_fit();    // reallocate buffer, invalidates iterators!
    std::cout << v.capacity() << '\n'    // >= 1000

    v.clear(); // erase all elements
}
```

# Std::deque

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <deque>

void f()
{
    // similar interface as for vector
    std::deque<int> d;

    // but no reserve and capacity
    d.reserve(1000); // pre-allocate buffer with >= 1000 capacity
    std::cout << d.capacity() << '\n' // >= 1000

    // but O(1) operations on front too
    d.push_front(42);
    d.emplace_front(43);

    d.pop_front();

    d.resize(1000); // add 1000 default int: 0
    std::cout << d.size() << '\n' // 1000
}
```

# Std::list

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <list>

void f()
{
    // similar interface as for deque
    std::list<int> lst;

    lst.resize(1000);    // add 1000 default int: 0
    std::cout << d.size() << '\n'    // 1000

    // but no direct access to random element
    std::cout << lst[5] << lst.at(2) << '\n';

    // O(1) operations on front and end
    lst.push_front(42);
    lst.emplace_front(43);
    lst.push_back(42);
    lst.emplace_back(43);

    lst.pop_front();
    lst.pop_back();
}
```

# Std::list

```
#include <iterator>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <list>

void f(std::list<int> &other)
{
    std::list<int> lst; // similar interface as for deque

    lst.push_front(42);
    lst.emplace_back(43);

    auto it = find( lst.begin(), lst.end(), 42);
    insert( it+2, 44); // iterator is bidirectional not random access iterator

    // list specific operations
    lst.merge(other); // merge elements from other list (sorted)
    lst.splice( it, other); // move elements from other list

    lst.reverse(); // reverse elements
    lst.sort(); // sort elements

    lst.remove(42); // removes all 42, not the algorithm, really erase elements
    lst.unique(42); // removes consecutive duplicate elements, erases too
}
```

# Std::forward\_list

```
template <typename T, typename Alloc = allocator<T> >
class forward_list
{
public:
    void clear();
    iterator insert_after(const_iterator pos, const T& value); // +move, +interval
    iterator emplace_after(const_iterator pos, Args&&... args);
    iterator erase_after(const_iterator pos); // +interval
    void push_front(const T& value); // +move
    void emplace_front( Args&&... args );
    void pop_front();
    void resize(size_type count);
    void resize(size_type count, const T& value);
    void swap(forward_list& other);
    bool empty();

    iterator before_begin(); // cbefore_begin()
    iterator begin(); // cbegin()
    iterator end(); // cend()

    void merge(forward_list&& other, Compare comp);
    void splice_after(const_iterator pos, forward_list& other);
    void remove(const T& value);
    void remove_if(UnaryPredicate p);
    void reverse();
    void unique(BinaryPredicate p);
    void sort(Compare comp); // sort()

    // no reverse iteration
    // no back() or push_back()
    // no size()
};
```

# Std::stack

```
#include <stack>

void f()
{
    // stack is a container adaptor
    // stack can be defined on anything having: back(), push_back(), pop_back()
    std::stack<int,vector> sv;    // stack over vector
    std::stack<int> st;         // stack is by default over deque

    // usual accessors: empty(), size()

    // stack interface
    st.push(42);                // at top
    st.emplace(43);            // at top

    auto val = st.top;         // read the top element
    st.top = 44;                // write the top element

    st.pop();                  // remove the top element

    // the underlying container is a protected member,
    // accessible if one inherits from stack
}
```

# Std::queue

```
#include <queue>

void f()
{
    // queue is a container adaptor
    // queue can be defined on anything having: back(), push_back(), pop_back()
    std::queue<int, list>  qv;    // queue over vector
    std::queue<int>  q;        // queue is by default over deque

    // usual accessors: empty(), size()

    // queue interface
    q.push(42);                // insert to the end
    q.emplace(43);            // create at the end

    auto first = q.front();    // read the first element
    auto last  = q.back();     // read the last element

    q.pop();                  // remove the first element

    // the underlying container is a protected member,
    // accessible if one inherits from stack
}
```

# Std::priority\_queue

```
#include <queue>

void f()
{
    // queue is a container adaptor
    // queue can be defined on anything having: back(), push_back(), pop_back()
    std::priority_queue<int, deque> pv;    // priority queue over deque
    std::priority_queue<int> pq;        // priority queue is by default over vector

    // usual accessors: empty(), size()

    // queue interface
    pq.push(42);    // insert into the priority_queue, keep heap invariant, O(log(n))
    pq.emplace(43); // create at the end then maintain heap invariant, O(log(n))

    auto first = pq.top();    // read the first element, O(1)
    pq.top() = 42;    // cannot write the top element
    pq.pop();    // remove the top element, keep heap invariant, O(log(n))

    // the underlying container is a protected member,
    // accessible if one inherits from stack
}
```

# Std::set

```
#include <set>

void f()
{
    // set is an associative container, ordered by key (the whole object by default)

    std::set<std::string> s;           // empty set of strings, comparator is std::less
    std::set<std::string,mycomp<std::string>> s2; // empty set of strings, custom comp.
    std::multiset<std::string> ms;    // empty set of strings, duplicates allowed

    // usual empty() size()
    s.insert(42); // insert into the set, keep ordering, O(log(n))
    s.insert(42); // insert the same key into the set fails, O(log(n))

    ms.insert(42); // insert into the multiset, keep ordering, O(log(n))
    ms.insert(42); // insert the same into the multiset works, neighbours, O(log(n))

    s.erase(42); // erase the only element with key 42 if any
    ms.erase(42); // erase one element with key 42 if any

    s.erase(s.begin()); // erase the first, smallest element
    ms.erase(--s.end()); // erase the last, largest element
}
```

# Std::map

```
#include <map>

void f()
{
    // map is an associative container of key+mapped type, ordered by key

    std::map<std::string, long, mycomp<std::string>> m; // map (string,long) custom comp.
    std::multimap<std::string> mm; // empty map of (string,long), key duplicates allowed

    // map specific interface
    m.insert({"Gabor"s, 12L}); // insert into map, keep ordering, O(log(n))
    m.insert({"Gabor"s, 13L}); // insert into map fails due key duplicate, O(log(n))
    mm.insert({"Gabor"s, 14L}); // insert into multimap ok, keep ordering, O(log(n))
    mm.insert({"Gabor"s, 15L}); // insert into multimap ok, keep ordering, O(log(n))

    // similar operations as set/multiset
    int n1 = m.count("Gabor"s); // 0 or 1 elements with the key
    bool b = m.contains("Gabor"s); // C++20
    auto it = m.find("Gabor"s); // return iterator to element 42 or end()
    auto [lb,ub] = mm.equal_range("Gabor"s); // return pair of iterator
    if ( lb != mm.end() )
        mm.erase(lb,ub); // erase all 42s
}
```

# Std::map

```
#include <map>

void f()
{
    // map is an associative container of key+mapped type, ordered by key

    std::map<std::string, long, mycomp<std::string>> m; // map (string, long) custom comp.

    m.insert({"Gabor"s, 12L}); // insert into map, keep ordering, O(log(n))
    m.insert({"Gabor"s, 13L}); // insert into map fails due key duplicate, O(log(n))

    // only on map (not even multimap!)
    long l1 = m.at("Gabor"s); // return reference to the mapped value
                                // or throw std::out_of_range
    m.at("Gabor"s) = 43L;

    long l2 = m["Gabor"s]; // return reference to the mapped value, if there where no
                            // such key, creates with default value of mapped type
    long l3 = m["Zoltan"s]; // maybe 0L

    m["Bjarne"s] = 42L; // insert or overwrite
}
```

# std::unordered\_map

```
#include <unordered_map>
#include <string>

using namespace std;

int main()
{
    unordered_map<string, string> hashtable;
    hashtable.emplace("www.zolix.hu", "212.92.23.158");
    hashtable.insert(make_pair("www.elte.hu", "212.92.23.159"));

    cout << "IP Address: " << hashtable["www.zolix.com"] << endl;

    for (auto &obj : hashtable)
    {
        cout << obj.first << ": " << obj.second << endl;
    }

    // returns std::unordered_map<std::string,double>::const_iterator
    auto it = hashtable.find("www.elte.com");
    if (hashtable.end() != it) // hashtable.count("www.elte.com") > 0
    {
        cout << it->first << ": " << it->second << endl;
    }
    return 0;
}
```

# std::unordered\_map

```
#include <unordered_map>
#include <string>

class MyClass
{
    std::string name;
    int age;
public:
    Bool operator==(const MyClass& rhs) { ... } // should be reflexive
    // ...
};

class MyClassHash
{
public:
    size_t operator()(const MyClass& m) const
    {
        return std::hash<std::string>()(m.name) ^ hash<int>()(m.age);
    }
};

int main()
{
    MyClass jim(...), joe(...);
    unordered_map<MyClass, double> salary;
    hashtable.emplace( jim, 20000);
    hashtable.emplace( joe, 22000);
    // ...
}
```

# Load factor

- Average insert/find is constant
- Worst-case: linear in container size
- Iterators are invalidated only on rehash
- Load factor == `size() / bucket_count()` // default 1.0
- Control of buckets:
  - `size_type bucket_count() const; // #of buckets`
  - `float max_load_factor() const; // get max load factor`
  - `void max_load_factor(float z); // set max load factor`
  - `size_type bucket_size ( size_type n ) const; // #of bucket n`
  - `size_type bucket( const key_t& key) const; // where key goes?`
  - `void rehash( size_type n); // sets #of buckets`

# std::unordered\_map

```
// unordered_map::bucket_count
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
    std::unordered_map<std::string, std::string> mymap = {
        {"house", "maison"}, {"apple", "pomme"}, {"tree", "arbre"},
        {"book", "livre"}, {"door", "porte"}, {"grapefruit", "pamplemousse"}
    };
    unsigned n = mymap.bucket_count();
    std::cout << "mymap has " << n << " buckets.\n";

    for (unsigned i=0; i<n; ++i) {
        std::cout << "bucket #" << i << " contains: ";
        for (auto it = mymap.begin(i); it!=mymap.end(i); ++it)
            std::cout << "[" << it->first << ":" << it->second << "] ";
        std::cout << "\n";
    }
}
mymap has 7 buckets.
bucket #0 contains: [book:livre] [house:maison]
bucket #1 contains:
bucket #2 contains:
bucket #3 contains: [grapefruit:pamplemousse] [tree:arbre]
bucket #4 contains:
bucket #5 contains: [apple:pomme]
bucket #6 contains: [door:porte]
```

# Load factor

```
// unordered_map::max_load_factor
int main ()
{
    std::unordered_map<std::string, std::string> mymap = {
        {"Au", "gold"}, {"Ag", "Silver"}, {"Cu", "Copper"}, {"Pt", "Platinum"}
    };

    std::cout << "current max_load_factor: " << mymap.max_load_factor() << std::endl;
    std::cout << "current size: " << mymap.size() << std::endl;
    std::cout << "current bucket_count: " << mymap.bucket_count() << std::endl;
    std::cout << "current load_factor: " << mymap.load_factor() << std::endl;

    float z = mymap.max_load_factor();
    mymap.max_load_factor ( z / 2.0 );

    std::cout << "new max_load_factor: " << mymap.max_load_factor() << std::endl;
    std::cout << "new size: " << mymap.size() << std::endl;
    std::cout << "new bucket_count: " << mymap.bucket_count() << std::endl;
    std::cout << "new load_factor: " << mymap.load_factor() << std::endl;
    return 0;
}
current max_load_factor: 1
current size: 4
current bucket_count: 5
current load_factor: 0.8
new max_load_factor: 0.5
new size: 4
new bucket_count: 11
new load_factor: 0.363636
```

# Std::unordered\_multi...

```
// unordered_multiset::equal_range

#include <iostream>
#include <string>
#include <unordered_set>

int main ()
{
    std::unordered_multiset<std::string> myums =
        {"cow", "pig", "pig", "chicken", "pig", "chicken"};

    auto myrange = myums.equal_range("pig");

    std::cout << "These pigs were found:";

    while ( myrange.first != myrange.second ) {
        std::cout << " " << *myrange.first++;
    }
    std::cout << std::endl;

    return 0;
}
```

# Example: merge two files

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()    // simple merge:
{
    string s1, s2;
    ifstream f1("file1.txt");
    ifstream f2("file2.txt");

    f1 >> s1; f2 >> s2;
    while (f1 || f2)
    {
        if (f1 && ((s1 <= s2) || !f2))
        {
            cout << s1 << endl;
            f1 >> s1;
        }
        if (f2 && ((s1 >= s2) || !f1))
        {
            cout << s2 << endl;
            f2 >> s2;
        }
    }
    return 0;
}
```

# Example: naïve STL usage

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>    // merge( b1, e1, b2, e2, b3 [,opc_rend])
#include <vector>

using namespace std;
int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    string s;
    vector<string> v1;
    while ( if1 >> s ) v1.push_back(s); // do we have enough memory?
    vector<string> v2;
    while ( if2 >> s ) v2.push_back(s); // do we have enough memory?

    // allocate the elements for the result, do we have enough memory?
    vector<string> v3(v1.size() + v2.size()); // constructs empty strings...

    // *b3++ = *b1 < *b2 ? *b1++ : *b2++
    merge( v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());

    for ( int i = 0; i < v3.size(); ++i)
        cout << v3[i] << endl;

    return 0;
}
```

# std::merge

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2) {
            return std::copy(first1, last1, d_first);
        }
        if (*first2 < *first1) {
            *d_first = *first2;
            ++first2;
        } else {
            *d_first = *first1;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

# std::merge

```
template<class InputIt1, class InputIt2, class OutputIt>
OutputIt merge(InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2,
               OutputIt d_first, Compare comp)
{
    for (; first1 != last1; ++d_first) {
        if (first2 == last2) {
            return std::copy(first1, last1, d_first);
        }
        if ( comp(*first2,*first1) ) {
            *d_first = *first2;
            ++first2;
        } else {
            *d_first = *first1;
            ++first1;
        }
    }
    return std::copy(first2, last2, d_first);
}
```

# Example: inserters

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

using namespace std;

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    string s;
    vector<string> v1;
    while ( if1 >> s ) v1.push_back(s);
    vector<string> v2;
    while ( if2 >> s ) v2.push_back(s);
    vector<string> v3;
    v3.reserve( v1.size() + v2.size() ); // alloc buffer but do not construct, size == 0

    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(v3)); // v3.push_back(*c)

    for ( int i = 0; i < v3.size(); ++i)
        cout << v3[i] << endl;

    return 0;
}
```

# Example: inserters

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

```
using namespace std;
```

```
int main()
```

```
{
```

```
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");
```

```
    string s;
```

```
    vector<string> v1;
```

```
    while ( if1 >> s ) v1.push_back(s);
```

```
    vector<string> v2;
```

```
    while ( if2 >> s ) v2.push_back(s);
```

```
    vector<string> v3;
```

```
    v3.reserve( v1.size() + v2.size() ); // alloc buffer but do not construct, size == 0
```

```
    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), back_inserter(v3)); // v3.push_back(*c)
```

```
    for ( int i = 0; i < v3.size(); ++i)
```

```
        cout << v3[i] << endl;
```

```
    return 0;
```

```
}
```

```
template<typename _Cont>
```

```
class back_insert_iterator : public
```

```
    iterator<output_iterator_tag, void, void, void, void>
```

```
{
```

```
    back_insert_iterator&
```

```
operator=(const typename _Cont::value_type& __value)
```

```
{ // same overloaded to reference and rvalue ref
    container->push_back(__value);
```

```
    return *this;
```

```
}
```

```
back_insert_iterator& operator*() { return *this; }
```

```
back_insert_iterator& operator++() { return *this; }
```

```
back_insert_iterator operator++(int){ return *this; }
```

```
protected:
```

```
    _Cont* container;
```

```
};
```

# Example: stream iterators

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>      // input- and output-iterators

using namespace std;

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    // istream_iterator(if1) -> if1 >> *current
    // istream_iterator() -> EOF
    // ostream_iterator(of,x) -> of << *current << x
    merge( istream_iterator<string>(if1), istream_iterator<string>(),
           istream_iterator<string>(if2), istream_iterator<string>(),
           ostream_iterator<string>(cout, "\n") );

    return 0;
}
```

# Example: comparator

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <iterator>

struct my_less // function object: "functor"
{
    bool operator()(const std::string& s1, const std::string& s2) {
        std::string us1 = s1;
        std::string us2 = s2;
        transform( s1.begin(), s1.end(), us1.begin(), toupper); // TODO: use <locale>
        transform( s2.begin(), s2.end(), us2.begin(), toupper);
        return us1 < us2;
    }
};

int main()
{
    ifstream if1("file1.txt");
    ifstream if2("file2.txt");

    merge( istream_iterator<string>(if1), istream_iterator<string>(),
           istream_iterator<string>(if2), istream_iterator<string>(),
           ostream_iterator<string>(cout, "\n"), my_less() );
    return 0;
}
```

# Example: template comparator

```
template <typename T>
class distr
{
public:
    distr(int l, int r, bool fl = true) : left_(l), right_(r), from_left_(fl), cnt_(0) { }
    // formal reasons: "compare" has two parameters of type T
    bool operator()( const T&, const T& ) {
        bool ret = from_left_; // from_left_ is "smaller" currently
        const int max = from_left_ ? left_ : right_;
        if ( ++cnt_ == max )
        {
            cnt_ = 0;
            from_left_ = ! from_left_;
        }
        return ret;
    }
private:
    const int left_; // read left_ element from left
    const int right_; // read right_ element from right
    int from_left_; // start from left
    int cnt_;
};

// ...
istream_iterator<string>(if2), istream_iterator<string>(),
ostream_iterator<string>(cout, "\n"), distr<std::string>(left, right) );
// ...
```

# Vector vs Associative containers

```
#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main() /* print unique sorted elems */
{
    set<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    copy( coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main() /* print unique sorted elems */
{
    vector<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    sort (coll.begin(), coll.end()); // sort elements
    unique_copy (coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}
```

# Vector vs Associative containers

```
#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

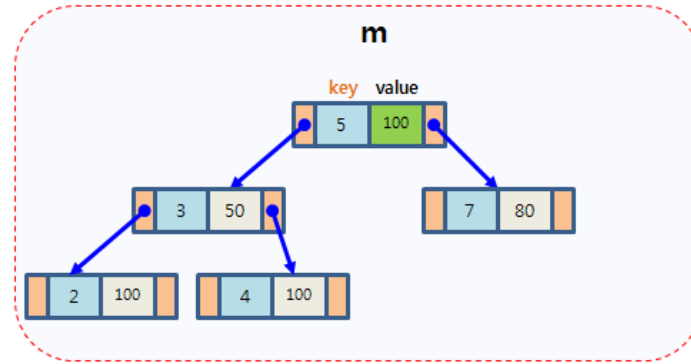
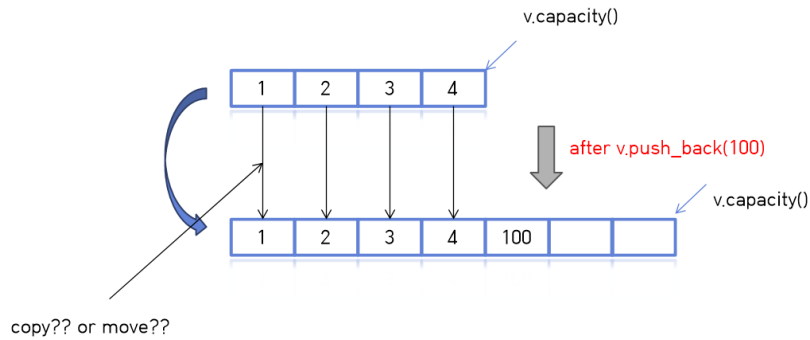
int main() /* print unique sorted elems */
{
    set<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    copy( coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

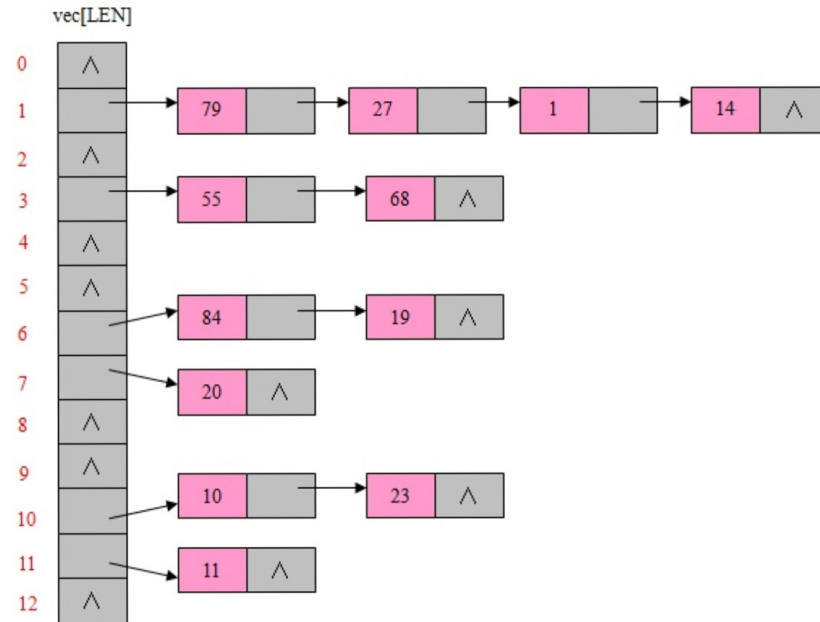
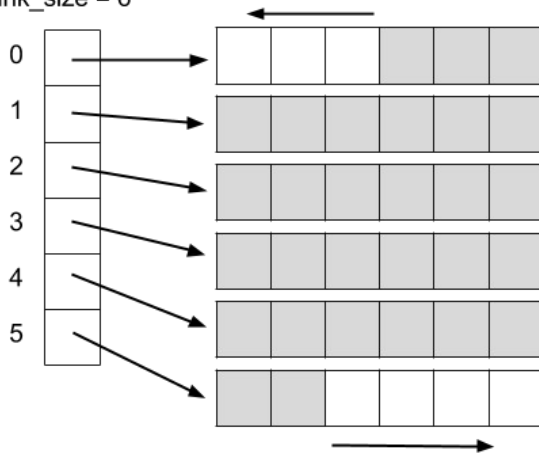
int main() /* print unique sorted elems */
{
    vector<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    sort (coll.begin(), coll.end()); // sort elements
    unique_copy (coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}

// 150.000 string: vector solution is better with 10%
// + reserve: 15%
// multiset + copy: 40%
```

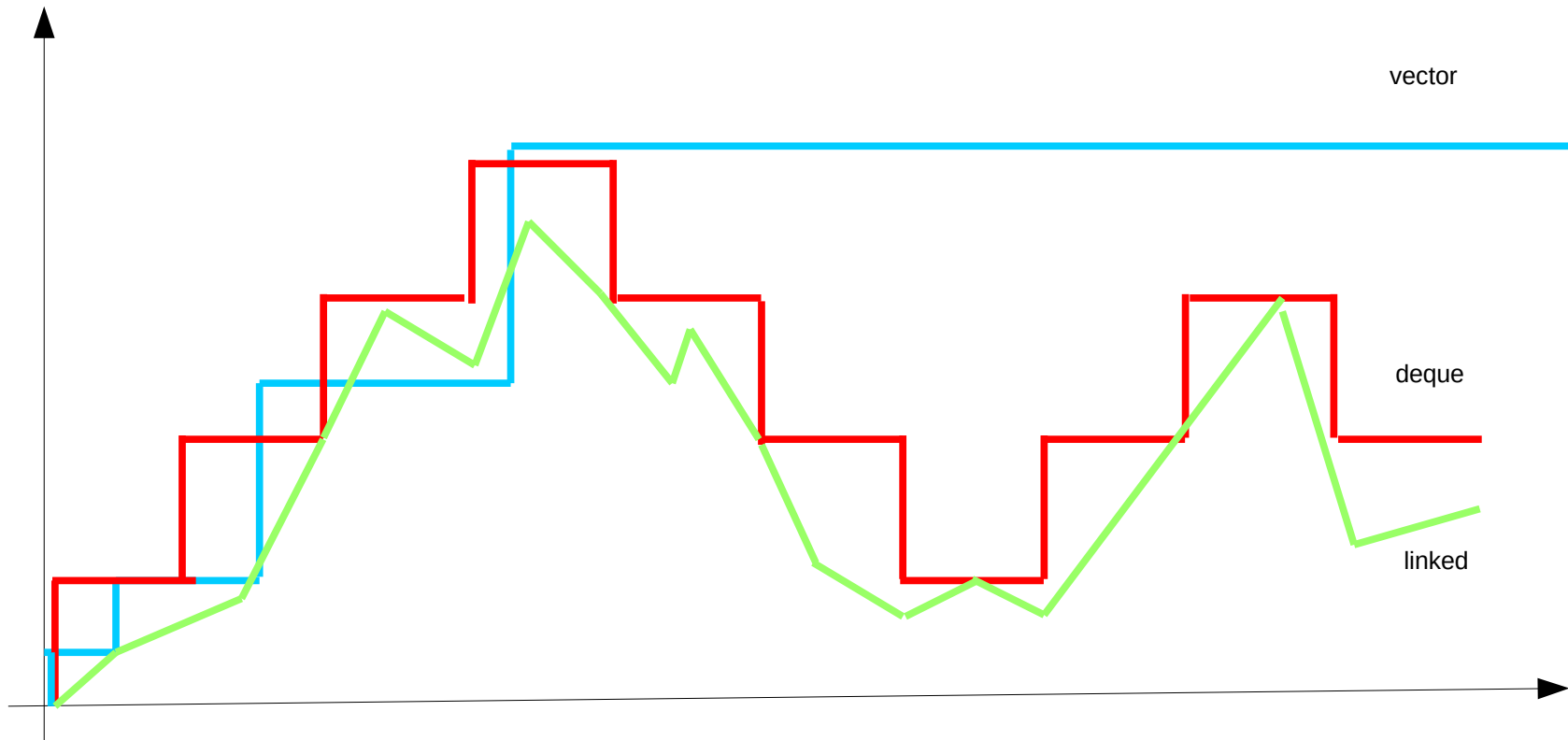
# Typical container implementations



shift = 3  
chunk\_size = 6



# Memory consumption



# Vector size and capacity

```
int main()
{
    std::vector<int> v;
    std::cout << "Default-constructed capacity is " << v.capacity() << '\n';

    v.resize(100);
    std::cout << "Capacity of a 100-element vector is " << v.capacity() << '\n';

    v.clear();
    std::cout << "Capacity after clear() is " << v.capacity() << '\n';

    // std::vector<int>(v).swap(v); // C++98

    v.shrink_to_fit(); // C++11
    std::cout << "Capacity after shrink_to_fit() is " << v.capacity() << '\n';
}
```

```
Default-constructed capacity is 0
Capacity of a 100-element vector is 100
Capacity after clear() is 100
Capacity after shrink_to_fit() is 0
```

# Iterator invalidation

| Category                         | Container                                                                              | After <b>insertion</b> , are... |                               | After <b>erasure</b> , are... |                      | Conditionally                                                                |
|----------------------------------|----------------------------------------------------------------------------------------|---------------------------------|-------------------------------|-------------------------------|----------------------|------------------------------------------------------------------------------|
|                                  |                                                                                        | iterators valid?                | references valid?             | iterators valid?              | references valid?    |                                                                              |
| Sequence containers              | <code>array</code>                                                                     | N/A                             |                               | N/A                           |                      |                                                                              |
|                                  | <code>vector</code>                                                                    | No                              |                               | N/A                           |                      | Insertion changed capacity                                                   |
|                                  |                                                                                        | Yes                             |                               | Yes                           |                      | Before modified element(s)<br>(for insertion only if capacity didn't change) |
|                                  | <code>deque</code>                                                                     | No                              |                               | No                            |                      | At or after modified element(s)                                              |
|                                  |                                                                                        | No                              | Yes                           | Yes, except erased element(s) |                      | Modified first or last element                                               |
|                                  | No                                                                                     |                                 | No                            |                               | Modified middle only |                                                                              |
| <code>list</code>                | Yes                                                                                    |                                 | Yes, except erased element(s) |                               |                      |                                                                              |
| <code>forward_list</code>        | Yes                                                                                    |                                 | Yes, except erased element(s) |                               |                      |                                                                              |
| Associative containers           | <code>set</code><br><code>multiset</code><br><code>map</code><br><code>multimap</code> | Yes                             |                               | Yes, except erased element(s) |                      |                                                                              |
| Unordered associative containers | <code>unordered_set</code><br><code>unordered_multiset</code>                          | No                              | Yes                           | N/A                           |                      | Insertion caused rehash                                                      |
|                                  | <code>unordered_map</code><br><code>unordered_multimap</code>                          | Yes                             |                               | Yes, except erased element(s) |                      | No rehash                                                                    |

From: <https://en.cppreference.com/w/cpp/container>

# STL traps

```
void f(std::vector<int> &v; int del)
{
    std::remove( v.begin(), v.end(), del);    // remove elements equal del ?
}
```

# STL traps

```
void f(std::vector<int> &v; int del)
{
    std::remove( v.begin(), v.end(), del);    // remove elements equal del ?
}
```

```
void f(std::vector<int> &v; int del)
{
    auto it = std::remove( v.begin(), v.end(), del);
    v.erase( it, v.end() );                    // remove elements equal del
}
```

# STL traps

```
void f(std::vector<int> &v)
{
    std::unique( v.begin(), v.end(), del);    // remove duplicates ?
}
```

# STL traps

```
void f(std::vector<int> &v)
{
    std::unique( v.begin(), v.end(), del);    // remove duplicates ?
}
```

```
void f(std::vector<int> &v; int del)
{
    auto it = std::unique( v.begin(), v.end());
    v.erase( it, v.end() );                // remove duplicates
}
```

# STL traps

```
void f(std::vector<int> &v)
{
    std::unique( v.begin(), v.end(), del);    // remove duplicates ?
}
```

```
void f(std::vector<int> &v; int del)
{
    std::sort( v.begin(), v.end());          // unique requires sorted input
    auto it = std::unique( v.begin(), v.end());
    v.erase( it, v.end() );                 // remove duplicates
}
```

# STL traps

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m[i] == key ? true : false;    // check whether contains key ?
}
```

# STL traps

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m[i] == key ? true : false;    // check whether contains key ?
}
```

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m.end() != m.find(key) ? true : false; // check whether contains key
}
```

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m.count(key) == 1 ? true : false;    // check whether contains key
}
```

# STL traps

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m[i] == key ? true : false;    // check whether contains key ?
}
```

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m.end() != m.find(key); // check whether contains key
}
```

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m.count(key) == 1; // check whether contains key
}
```

# STL traps

```
bool f(std::map<std::string, int> &m, std::string key)
{
    return m[i] == key ? true : false;    // check whether contains key ?
}
```

```
bool f(std::multimap<std::string, int> &m, std::string key)
{
    return m.end() != m.find(key); // check whether contains key
}
```

```
bool f(std::multimap<std::string, int> &m, std::string key)
{
    return m.count(key) > 0; // check whether contains key
}
```

# STL traps

```
bool f(std::list<std::string> &lst, std::string key)
{
    for ( auto it = lst.begin(); it != lst.end(); ++it)
    {
        if ( *it == key )
            lst.erase(it);           // erase all elements equal to key ?
    }
}
```

# STL traps

```
bool f(std::list<std::string> &lst, std::string key)
{
    for ( auto it = lst.begin(); it != lst.end(); ++it)
    {
        if ( *it == key )
            lst.erase(it);           // erase all elements equal to key ?
    }
}

bool f(std::list<std::string> &lst, std::string key)
{
    for ( auto it = lst.begin(); it != lst.end(); )
    {
        if ( *it == key )
            lst.erase(it++);        // erase all elements equal to key
        else
            ++it;
    }
}
```

# STL traps

```
bool f(std::list<std::string> &lst, std::string key)
{
    for ( auto it = lst.begin(); it != lst.end(); ++it)
    {
        if ( *it == key )
            lst.erase(it);           // erase all elements equal to key ?
    }
}

bool f(std::list<std::string> &lst, std::string key)
{
    for ( auto it = lst.begin(); it != lst.end(); )
    {
        if ( *it == key )
            lst.erase(it++);        // erase all elements equal to key
        else
            ++it;
    }
}

bool f(std::list<std::string> &lst, std::string key)
{
    auto it = std::remove(lst.begin(), lst.end(), key);
    lst.erase( it, lst.end() );    // erase all elements equal to key
}
```

# STL traps

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

```
std::string f(std::vector<std::string> &v1, std::vector<std::string> &v2, int i)
{
    std::vector<std::vector> v3;
    v3.resize( v1.size()+v2.size() );

    std::merge( v1.begin(), v2.end(), v2.begin(), v1.end(), v3.begin());
    return v3.at(i); // return the i.th element of the merged vectors
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Segmentation fault (core dumped)
```

# STL traps

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

```
std::string f(std::vector<std::string> &v1, std::vector<std::string> &v2, int i)
{
    std::vector<std::vector> v3;
    v3.resize( v1.size()+v2.size() );

    std::merge( v1.begin(), v2.end(), v2.begin(), v1.end(), v3.begin());
    return v3.at(i);    // return the i.th lement of the merged vectors
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Segmentation fault (core dumped)
```

# Ranges

- Ranges are representing an iterable sequence
- In form of [first, sentinel]
  - Containers: [first, last)
  - Arrays: [first, size)
  - Streams: [first, predicate) (EOF)
  - Generators: [first, ...)
- Range algorithms
  - Applied eagerly
- Range adaptors
  - Applied to views lazily
  - Can be composed

# Ranges

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

```
std::string f(std::vector<std::string> &v1, std::vector<std::string> &v2, int i)
{
    std::vector<std::string> v3;
    v3.resize( v1.size()+v2.size() );

    std::ranges::merge( v1, v2, v3.begin());
    return v3.at(i); // return the i.th lement of the merged vectors
}
int main()
{
    std::vector<std::string> v1={"Hello", "world"};
    std::vector<std::string> v2={"Hallo", "welt"};
    std::cout << f(v1,v2,1);

    return 0;
}
$ g++ -Wextra merge.cpp
$ ./a.out
Hello
```

# Ranges

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

```
int main()
{
    std::vector<std::string> v1={"Hello", "world", "Hallo", "welt"};

    std::sort( v1.begin(), v1.end());

    for ( auto s: v1 )
        std::cout << s << ' ';
    std::cout << '\n';

    return 0;
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Hallo Hello welt world
```

# Ranges

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

```
int main()
{
    std::vector<std::string> v1={"Hello", "world", "Hallo", "welt"};

    std::ranges::sort(v1);    //std::sort( v1.begin(), v1.end());

    for ( auto s: v1 | std::views::take(3) )
        std::cout << s << ' ';
    std::cout << '\n';

    return 0;
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Hallo Hello welt
```

# Ranges

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

```
int main()
{
    std::vector<std::string> v1={"Hello", "world", "Hallo", "welt"};
    auto len5 = [](std::string s) { return s.length() == 5; };

    std::ranges::sort( v1);

    for ( auto s: v1 | std::views::filter(len5) )
        std::cout << s << ' ';
    std::cout << '\n';

    return 0;
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Hallo Hello world
```

# Ranges

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

int main()
{
    std::vector<std::string> v1={"Hello", "world", "Hallo", "welt"};
    auto len5 = [](std::string s) { return s.length() == 5; };

    std::ranges::sort( v1);

    for ( auto s: v1 | std::views::drop(1)
          | std::views::take(3)
          | std::views::filter(len5) )
        std::cout << s << ' ';
    std::cout << '\n';

    return 0;
}
```

```
$ g++ -Wextra merge.cpp
$ ./a.out
Hello world
```

# Example: word counter

```
#include <cctype>
#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<char> v{};
    std::cin >> std::noskipws;
    std::copy( std::istream_iterator<char>{std::cin},
              std::istream_iterator<char>{}, std::back_inserter(v));
    int cnt = 0;
    char prev = '\n'; // the imaginary char on -1 position is a white space.

    for ( char curr : v )
    {
        if ( std::isspace(prev) && !std::isspace(curr) ) // new word starts
            ++cnt;
        prev = curr;
    }
    std::cout << cnt << '\n';
}
```

# Example: word counter

```
#include <cctype>
#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

int main()
{
    std::vector<char> v{};
    std::cin >> std::noskipws;
    std::copy( std::istream_iterator<char>{std::cin},
               std::istream_iterator<char>{}, std::back_inserter(v));
    int cnt = 0;
    if ( ! v.empty() )
    {
        cnt = std::transform_reduce(
            std::begin(v), std::end(v)-1, std::begin(v)+1, !std::isspace(v[0])?1:0, std::plus{},
            [](<char> curr, <char> next){ return std::isspace(curr) && !std::isspace(next); } );

        std::cout << cnt << '\n';
    }
}
```

# Example: word counter

```
#include <cctype>
#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <execution>

int main()
{
    std::vector<char> v{};
    std::cin >> std::noskipws;
    std::copy( std::istream_iterator<char>{std::cin},
              std::istream_iterator<char>{}, std::back_inserter(v));
    int cnt = 0;
    if ( ! v.empty() )
    {
        cnt = std::transform_reduce( std::execution::par,
                                     std::begin(v), std::end(v)-1, std::begin(v)+1, !std::isspace(v[0])?1:0, std::plus{},
                                     [](<char> curr, <char> next){ return std::isspace(curr) && !std::isspace(next); } );
    }

    std::cout << cnt << '\n';
}
```

# Example: word counter

```
#include <cctype>
#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <execution>

int main()
{
    std::vector<char> v{};
    std::cin >> std::noskipws;
    std::copy( std::istream_iterator<char>{std::cin},
              std::istream_iterator<char>{}, std::back_inserter(v));
    int cnt = 0;
    if ( ! v.empty() )
    {
        cnt = std::transform_reduce( std::execution::par,
            std::begin(v), std::end(v)-1, std::begin(v)+1, 0, std::plus{},
            [](<strong>char curr, char next){ <strong>return std::isspace(curr) && !std::isspace(next); } )
            + !std::isspace(v[0]?1:0);
    }

    std::cout << cnt << '\n';
}
```

# par\_algorithms (C++17)

- Based on Intel's Threading Building Blocks (TBB)
- Extends STL algorithms with execution policy
  - `std::execution::seq` Sequential execution
  - `std::execution::par` Parallel execution
  - `std::execution::par_unseq` Parallel SIMD execution
  - `std::execution::unseq` Sequential SIMD execution
- These policies are permissions not obligations. Implementation may choose what can be parallelized
- Minimal requirement: forward iterator
- The programmer's task to ensure that element access functions will not cause dead lock or data race
- In case of parallelization and vectorization access must not use any blocking synchronization

# Vectorization

```
std::vector<int> v {1,2, ... };
```

```
int sum { std::accumulate(v.begin(), v.end(), 0) };
```

```
int sum = 0;
for ( size_t i = 0; i < v.size(); ++i)
{
    sum += v[i];
}
```

```
int sum = 0;
for ( size_t i = 0; i < v.size() / 4; i+=4)
{
    sum += v[i] + v[i+1] + v[i+2] + v[i+3];    // most CPU supports this
}
// handle if (v.size()/4) is not 0
```

# Parallel STL

```
// Example from Stroustrup
```

```
template<class T, class V>
struct Accum // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v) // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

# Parallel STL

```
// Example from cppreference
```

```
template<class T, class V>
struct Accum // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};
```

```
double comp(vector<double>& v)
{
    // non-deterministic if binary_op is not associative or not commutative
    double res = std::reduce(std::execution::par, v.begin(), v.end(), 0.0);
    return res;
}
```

# Dynamic policy

```
std::size_t threshold= ...; // some value
```

```
template <class ForwardIt>
```

```
void quicksort(ForwardIt first, ForwardIt last)
```

```
{
```

```
    if(first == last) return;
```

```
    std::size_t distance= std::distance(first, last);
```

```
    auto pivot = *std::next(first, distance/2);
```

```
    std::parallel::execution_policy exec_pol = std::parallel::par;
```

```
    if ( distance < threshold ) exec_pol = std::parallel_execution::seq;
```

```
    ForwardIt middle1 = std::partition(exec_pol, first, last,  
                                     [pivot](const auto& em){ return em < pivot; });
```

```
    ForwardIt middle2 = std::partition(exec_pol, middle1, last,  
                                     [pivot](const auto& em){ return !(pivot < em); });
```

```
    quicksort(first, middle1);
```

```
    quicksort(middle2, last);
```

```
}
```

# Algorithms with execution policy

- `std::adjacent_difference`
- `std::adjacent_find`
- `std::all_of`
- `std::any_of`
- `std::copy`
- `std::copy_if`
- `std::copy_n`
- `std::count`
- `std::count_if`
- `std::equal`
- `std::fill`
- `std::fill_n`
- `std::find`
- `std::find_end`
- `std::find_first_of`
- `std::find_if`
- `std::find_if_not`
- `std::generate`
- `std::generate_n`
- `std::includes`
- `std::inner_product`
- `std::inplace_merge`
- `std::is_heap`
- `std::is_heap_until`
- `std::is_partitioned`
- `std::is_sorted`
- `std::is_sorted_until`
- `std::lexicographical_compare`
- `std::max_element`
- `std::merge`
- `std::min_element`
- `std::minmax_element`
- `std::mismatch`
- `std::move`
- `std::none_of`
- `std::nth_element`
- `std::partial_sort`
- `std::partial_sort_copy`
- `std::partition`
- `std::partition_copy`
- `std::remove`
- `std::remove_copy`
- `std::remove_copy_if`
- `std::remove_if`
- `std::replace`
- `std::replace_copy`
- `std::replace_copy_if`
- `std::replace_if`
- `std::reverse`
- `std::reverse_copy`
- `std::rotate`
- `std::rotate_copy`
- `std::search`
- `std::search_n`
- `std::set_difference`
- `std::set_intersection`
- `std::set_symmetric_difference`
- `std::set_union`
- `std::sort`
- `std::stable_partition`
- `std::stable_sort`
- `std::swap_ranges`
- `std::transform`
- `std::uninitialized_copy`
- `std::uninitialized_copy_n`
- `std::uninitialized_fill`
- `std::uninitialized_fill_n`
- `std::unique`
- `std::unique_copy`

# ... and a few new algorithms

```
std::for_each  
std::for_each_n  
std::exclusive_scan  
std::inclusive_scan  
std::transform_exclusive_scan  
std::transform_inclusive_scan  
std::reduce  
std::transform_reduce
```

# Protection?

```
int numComp= 0;  
  
std::vector<int> vec={1, 3, 8, 9, 10};  
  
std::sort(std::executon::par, vec.begin(), vec.end(),  
          [&numComp](int fir, int sec){ numComp++; return fir < sec; });
```

# Protection?

```
int numComp= 0;  
  
std::vector<int> vec={1,3,8,9,10};  
  
std::sort(std::parallel::vec, vec.begin(), vec.end(),  
          [&numComp](int fir, int sec){ numComp++; return fir < sec; });
```

Race condition == Undefined behavior

# Protection?

```
int numComp= 0;  
  
std::vector<int> vec={1,3,8,9,10};  
  
std::sort(std::parallel::vec, vec.begin(), vec.end(),  
          [&numComp](int fir, int sec){ numComp++; return fir < sec; });
```

Race condition == Undefined behavior

Also: must not use blocking mutexes

# accumulate() vs reduce()

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<long long> v1;
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }

    long long sum = 0;
    for ( std::size_t i = 0; i < v1.size(); ++i) // summa x^2 x in [0..49]
    {
        sum += v1[i]*v1[i];
    }

    std::cout << sum << '\n';

    return 0;
}
```

```
$ ./a.out
300
```

# accumulate() vs reduce()

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

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }

    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum); // classical STL

    std::cout << sum1 << '\n';
    return 0;
}

$ ./a.out
300
```

# accumulate() vs reduce()

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

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4}); // creates 50 elements
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum); // classical STL

    std::cout << sum1 << '\n';
    return 0;
}

$ ./a.out
300
```

# accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 10; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}
```

```
$ ./a.out
300, 300
```

# accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}
```

```
$ ./a.out
30000, 30000
```

# accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}
```

```
$ ./a.out
30000000, 59820950156796
```

# accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>

std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; }; // not commutative

int main()
{
    for (int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    // reduce can work parallel
    auto sum2 = std::reduce(std::execution::par, v1.begin(), v1.end(), 0LL, sqrsum);

    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}
```

```
$ ./a.out
30000000, 59820950156796
```

# accumulate() vs reduce()

```
#include <iostream>
#include <numeric>
#include <vector>
#include <execution>
#include <functional>
std::vector<long long> v1;
auto sqrsum = [] (auto s, auto val) { return s + val * val; };

int main()
{
    for ( int i = 0; i < 1000000; ++i)
    {
        v1.insert( v1.end(), {0,1,2,3,4});
    }
    // accumulate is guaranteed left associative
    auto sum1 = std::accumulate(v1.begin(), v1.end(), 0LL, sqrsum);
    auto sum2 = std::transform_reduce(std::execution::par, // map-reduce
        v1.begin(), v1.end(), 0LL,
        std::plus<>(),
        [] (auto v) { return v*v; });
    std::cout << sum1 << ", " << sum2 << '\n';
    return 0;
}

$ ./a.out
30000000, 30000000
```