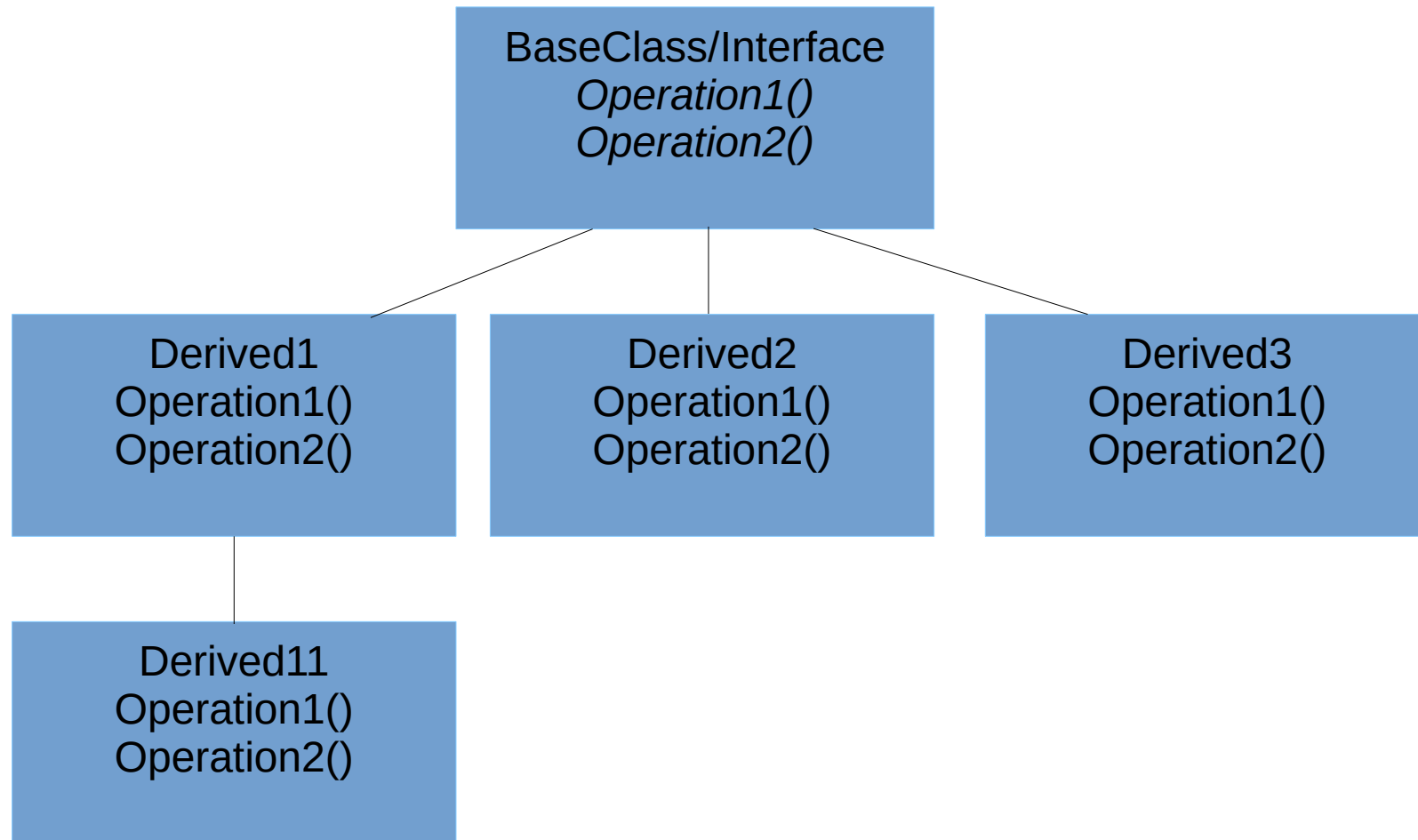


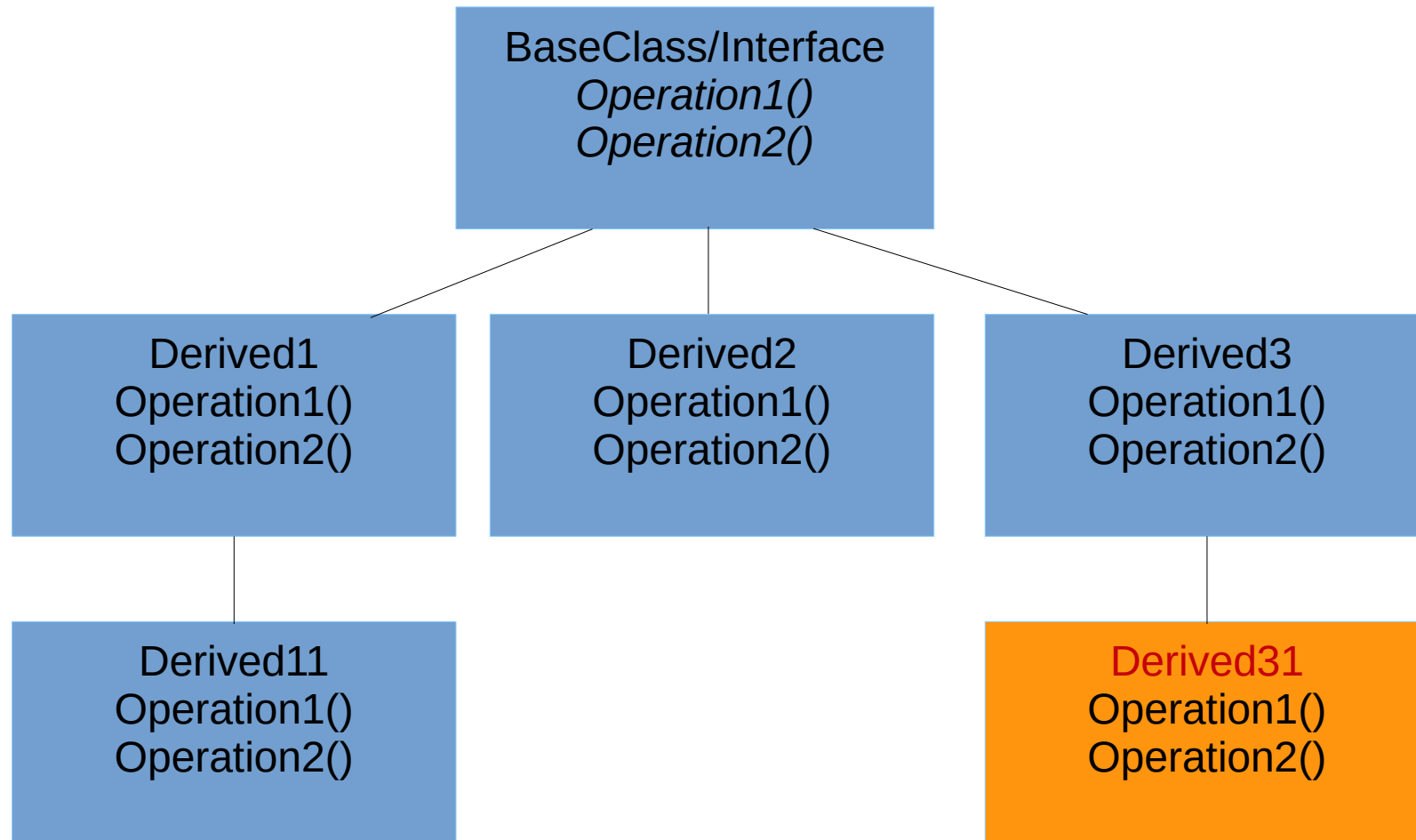
STL

- 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

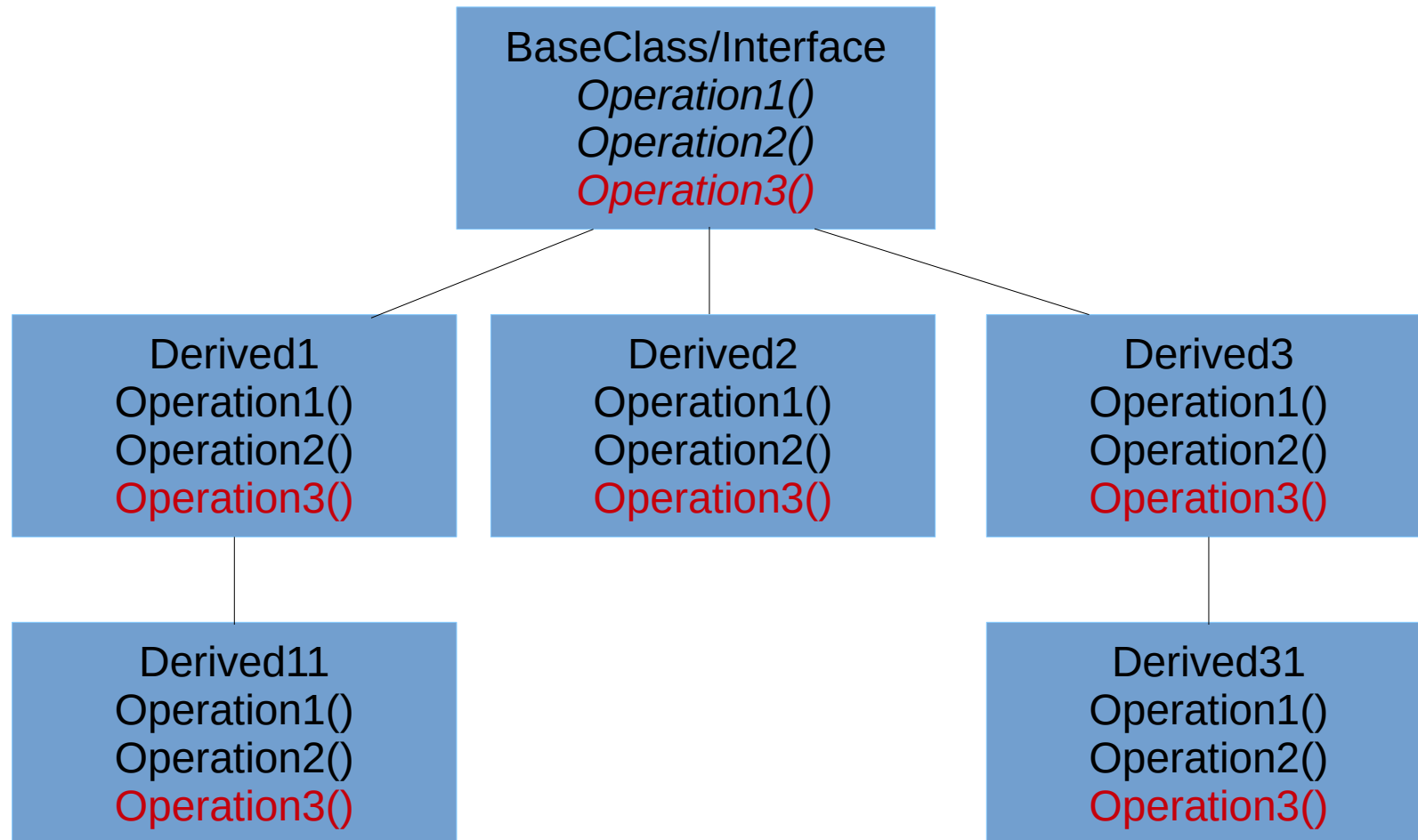
The expression problem



The expression problem



The expression problem

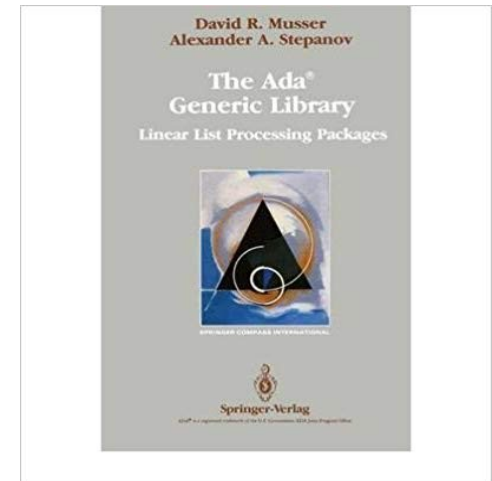
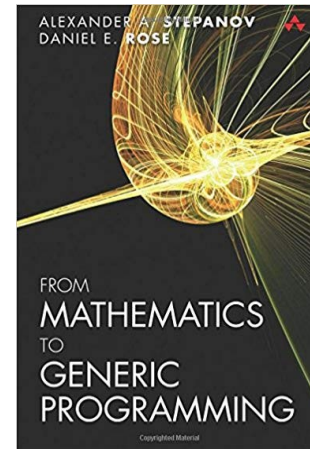
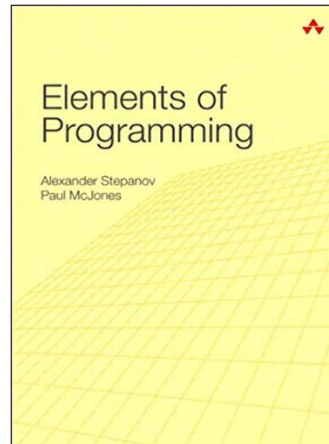


The expression problem

- 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

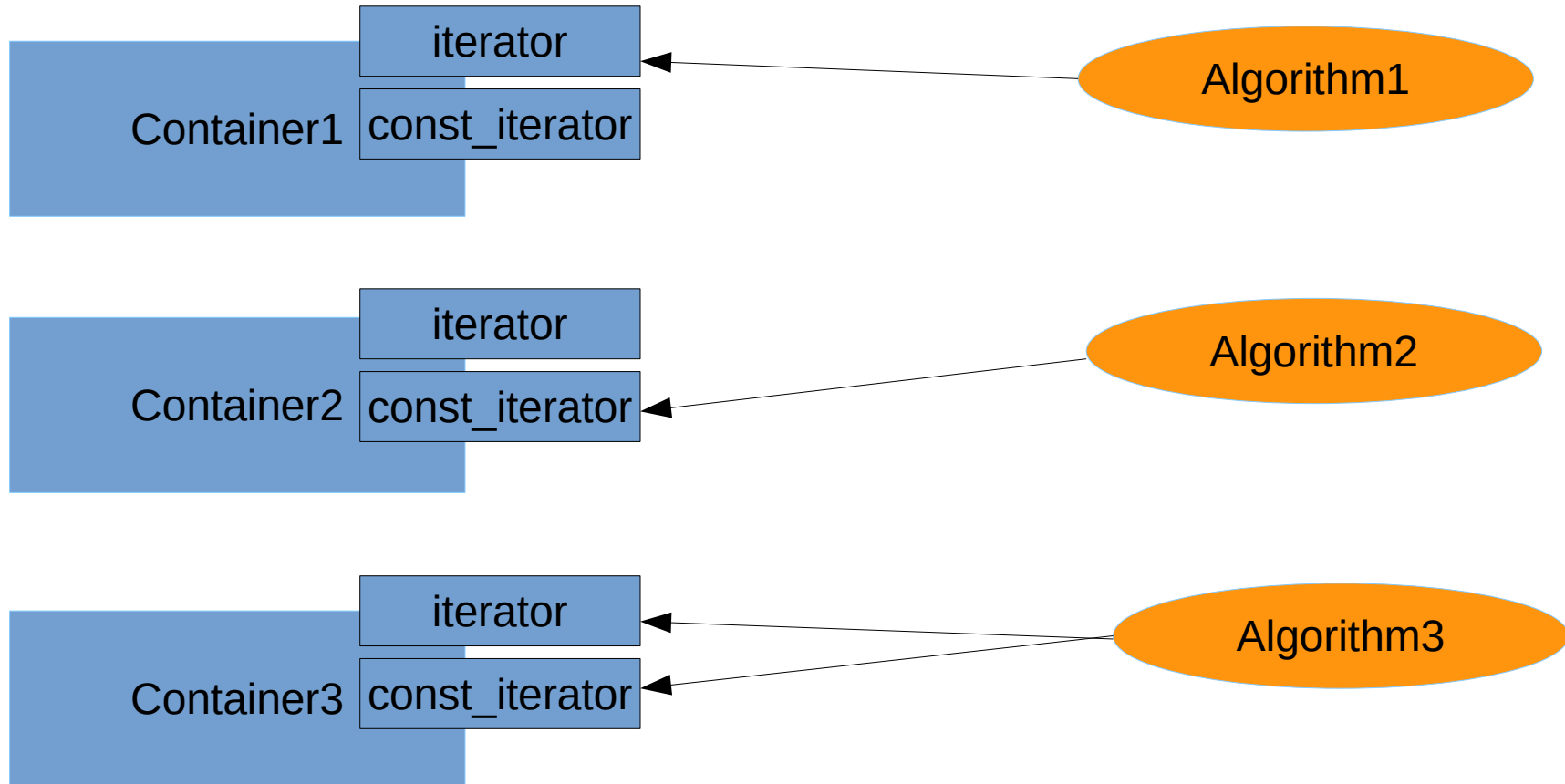
- Philip Wadler: expression problem mail, 1990
- Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman: "Synthesizing Object-Oriented and Functional Design to Promote Re-Use", 1998
- Aleksey Stepanov: generic programming, 1985



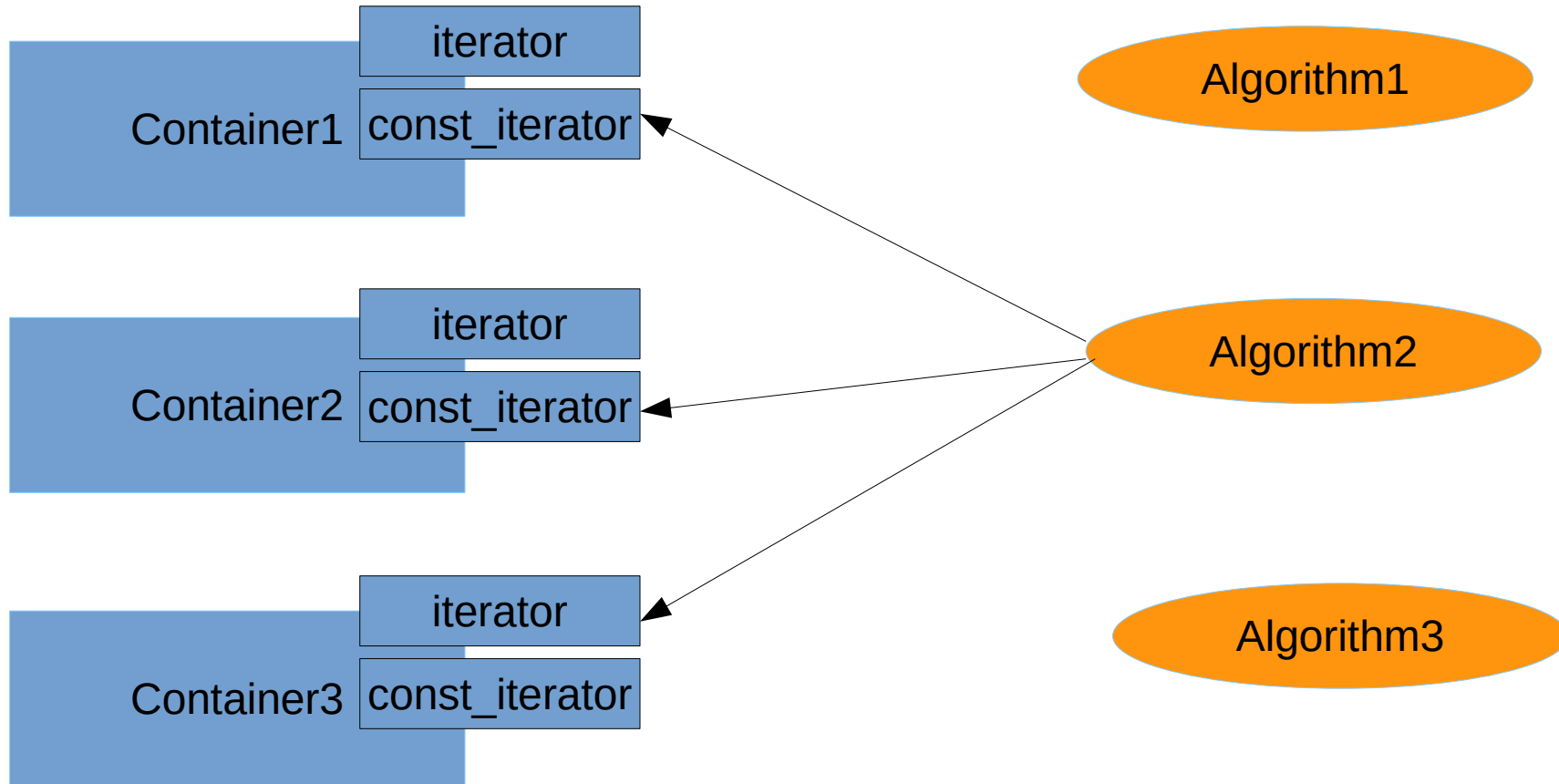
The STL solution



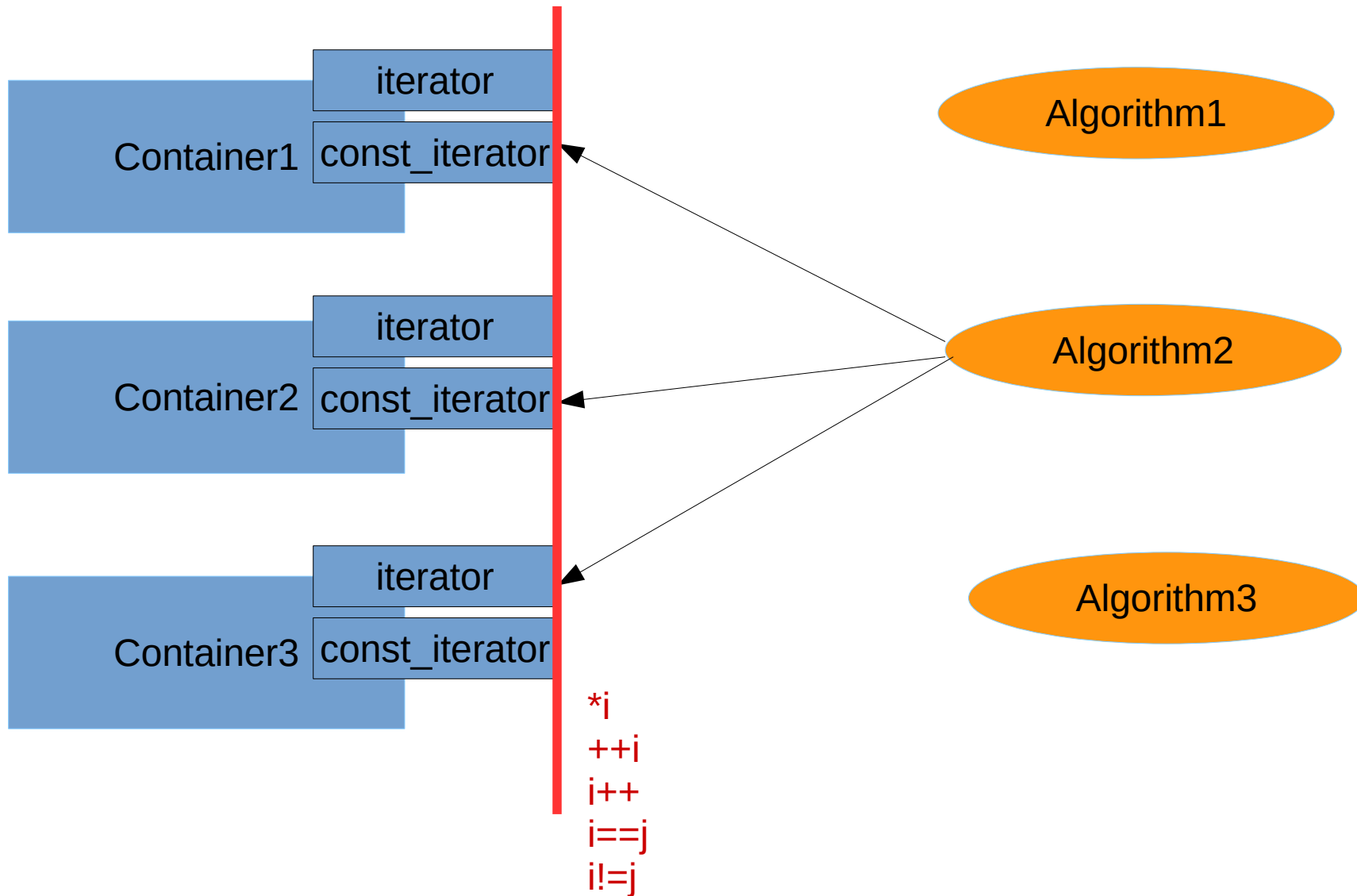
The STL solution



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() != i )
{
    *i = 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, ... };

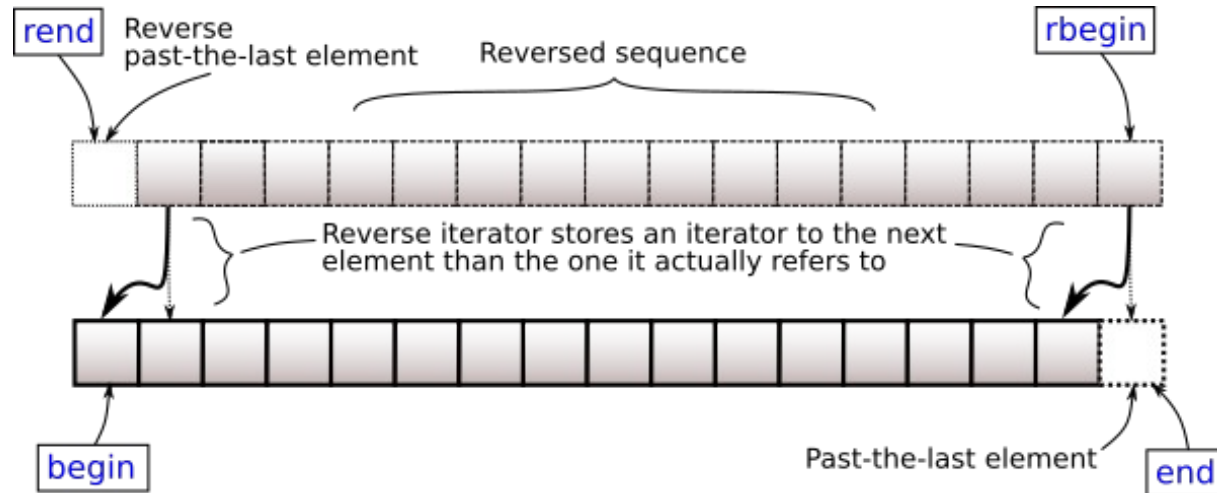
template <typename Container>
auto generic_find( const Container& c, typename Container::value_type& v)
{
    return std::find( c.begin(), c.end(), v);
}

template<class C>
typename C::value_type generic_sum(const C& c)
{
    typename C::value_type s{};
    typename C::const_iterator p = c.begin(); // auto p = c.begin()

    while ( p != c.end() )
    {
        s += *p;
        ++p;
    }
    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
```

Functors

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

// find the third occurrence less than value 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 less than value 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 less than value 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 less than value 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 less than value 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 less than value 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 less than value 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));
}
```

Example: merge two files

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

using namespace std;

// simple merge
int main()
{
    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

```
#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);
    vector<string> v2;
    while ( if2 >> s ) v2.push_back(s);

    // allocate the space for the result
    vector<string> v3(v1.size() + v2.size());    // very expensive...

    merge( v1.begin(),v1.end(),v2.begin(),v2.end(),v3.begin());    // v3[i] = *c

    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() );    // allocates but 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() );    // allocates but 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 iterator

```
#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;

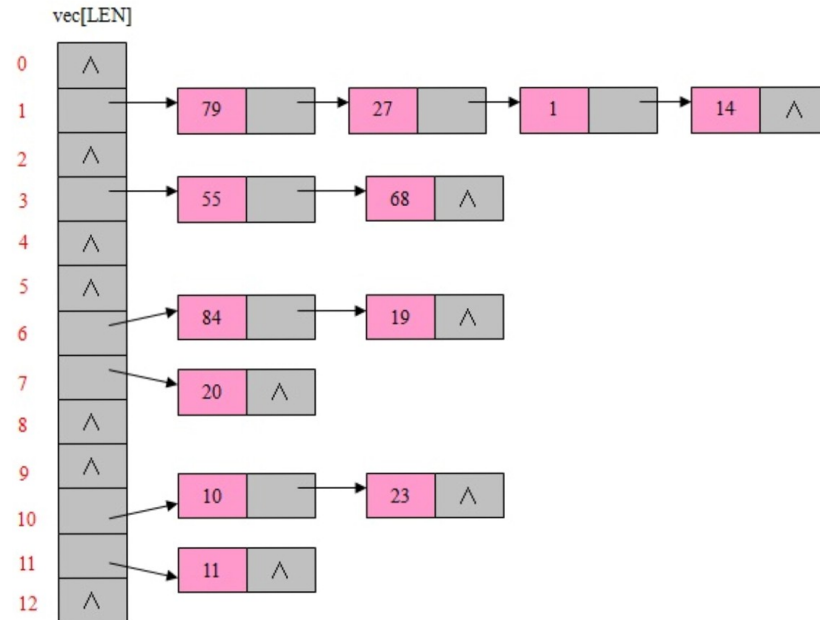
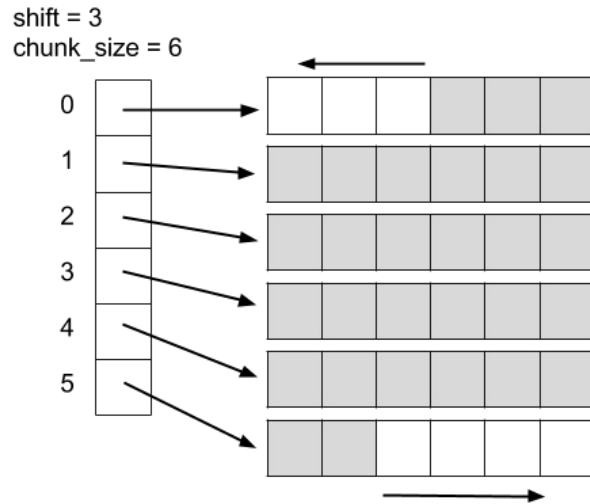
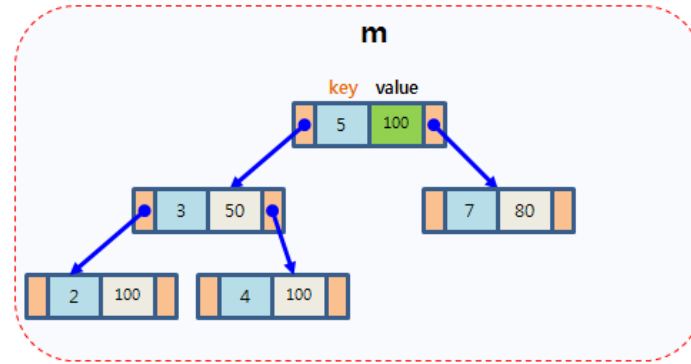
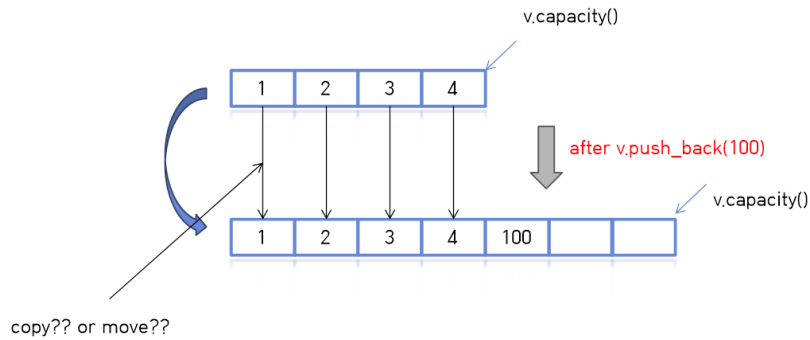
int main() /* print unique sorted elems */
{
    set<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    copy( coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}

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

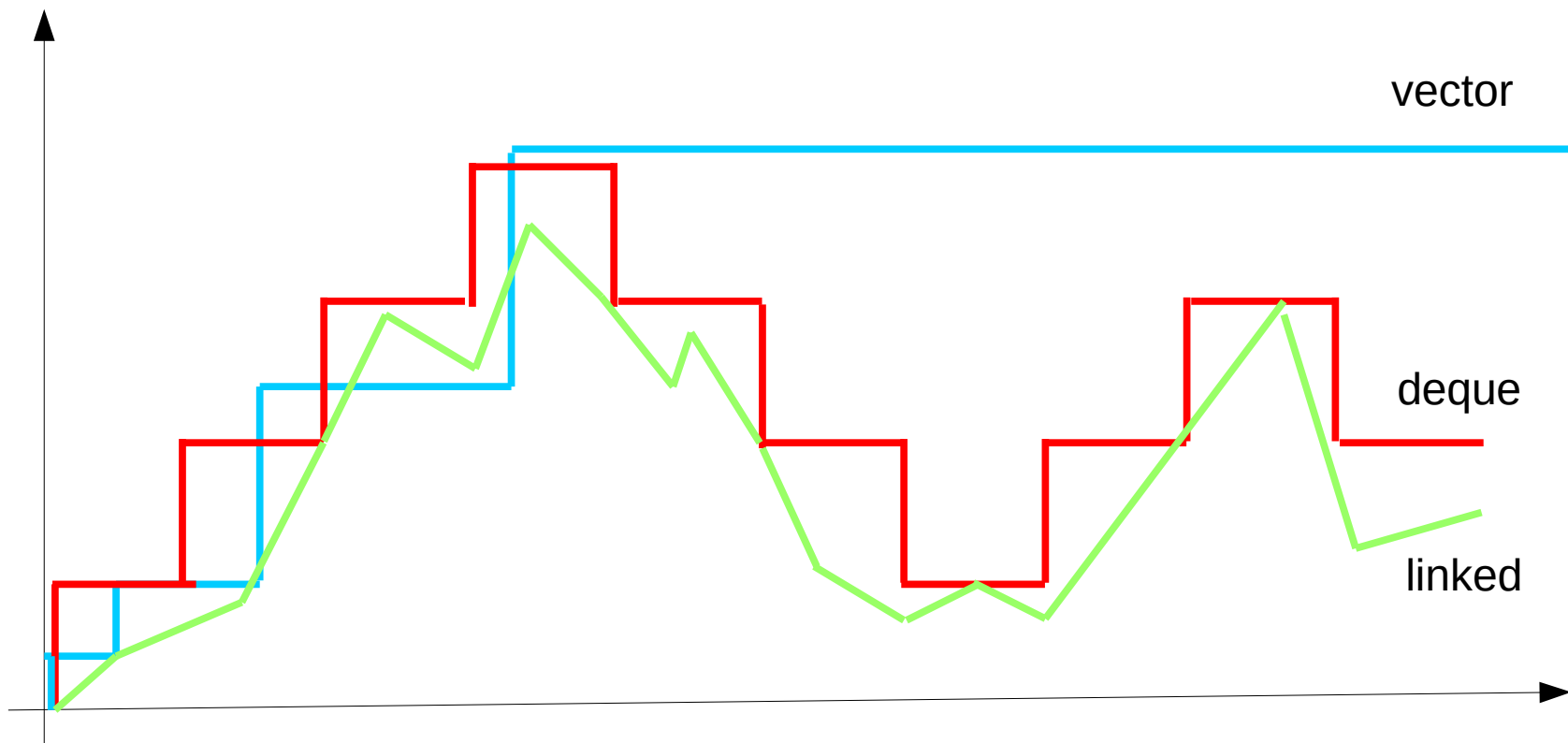
int main() /* print unique sorted elems */
{
    vector<string> coll( istream_iterator<string>(cin), istream_iterator<string>());
    sort (coll.begin(), coll.end()); // sort elements
    unique_copy (coll.begin(), coll.end(), ostream_iterator<string>(cout, "\n"));
}

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

Typical container implementations



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
```

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 << ' ';
}

// make_array and to_array are under consideration in standard
```


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);

    iterator before_begin(), begin(), end();
    bool empty();

    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);
    sort(Compare comp);

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

Hash-based containers

- `Unordered_map`
- `Unordered_set`
- `Unordered_multimap`
- `Unordered_multiset`

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;
}
```