Coping with Read-Only Set Iterators

SOFTWARE DEVELOPMENT CONFERENCE & EXPO

Build better software ...

Angelika Langer

Training / Consulting

http://www.AngelikaLanger.com

objective

- learn about the iterator type of the STL container `set`
- different implementations of the STL provide different iterators
  - read-only iterators
  - read-write iterators
- result: portability problem and other surprises

- see why set iterators are different from other container iterators
- identify rules for safe use of set iterators
- find work-arounds for restrictions of read-only set iterators

agenda

- mechanics of tree-based containers
- dangerous algorithms
- read-only vs. read-write set iterators
- iterator adapters

set

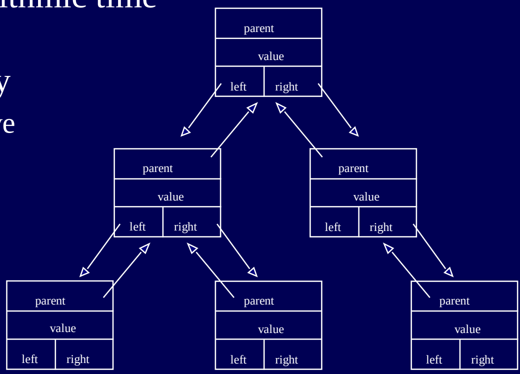
- STL container
- ordered collection of elements
  - needs a comparator

```
template <class Key, class Compare = less<Key> >
class set;
```

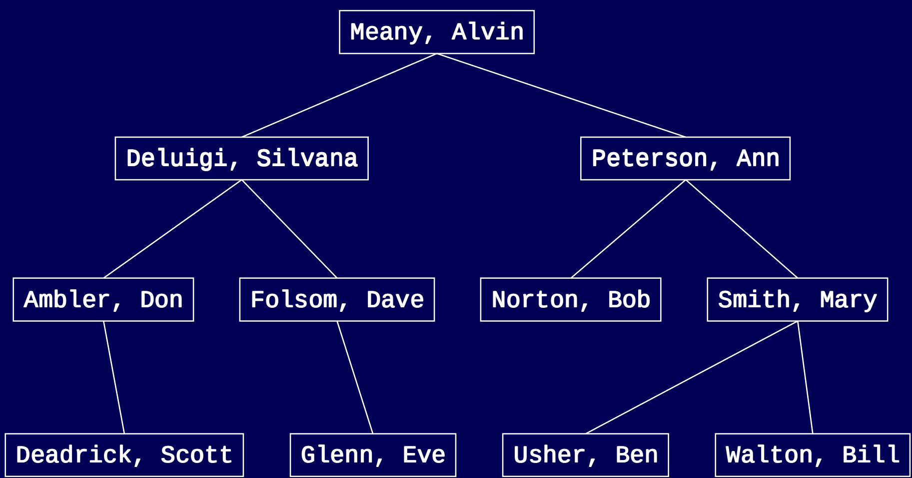
- based on a balanced binary tree
  - follows from complexity guarantees in the STL
  - logarithmic complexity for insertion, removal and search

binary tree

- elements maintained in sorting order
  - required sorting order and comparator
- left leaf is less than right leaf
- element access in logarithmic time
  - if tree is balanced
- re-balanced if necessary
  - during insert and remove



example: almost balanced binary tree



modification of set elements

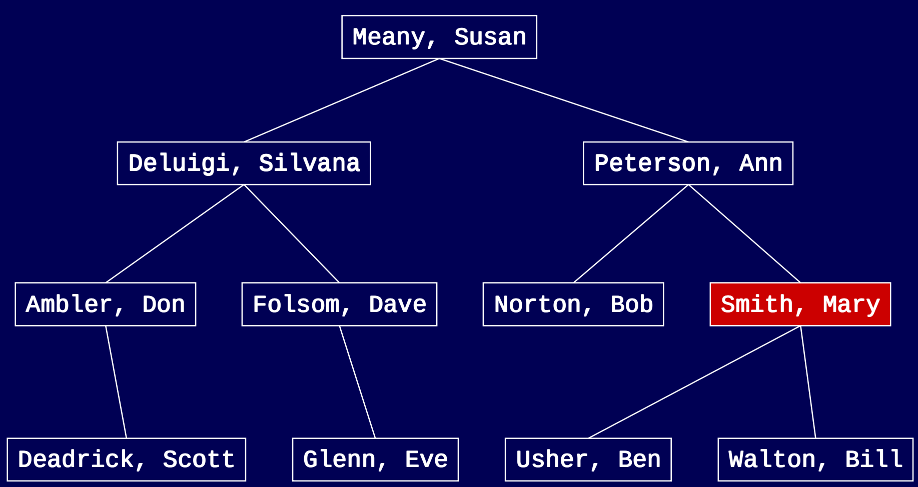
```
struct name {
 string _first, _last;
 ...
};
bool operator<(name lhs, name rhs)
{ return lhs._last < rhs._last; }
```

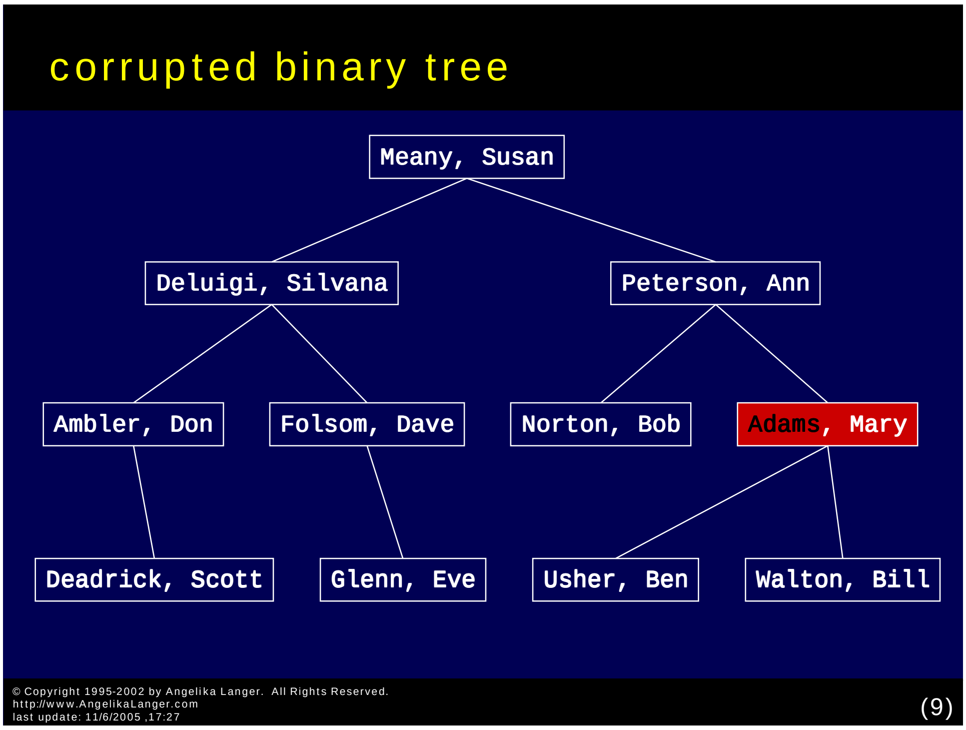
- modify element in `set` container
  - Mary Smith marries; modify last name

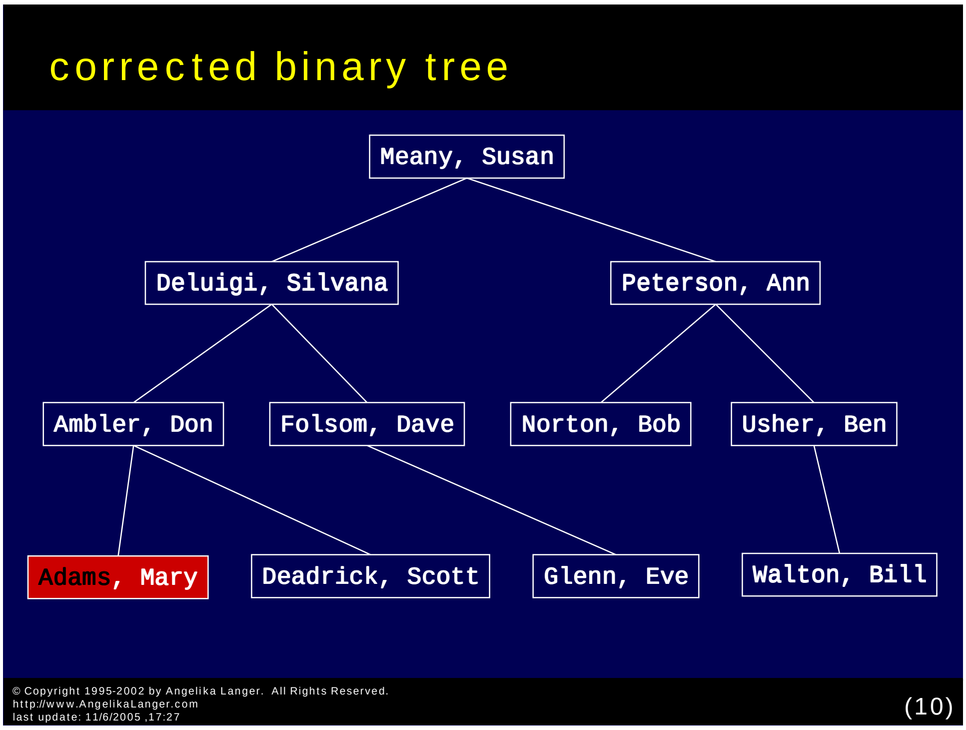
```
set<name> clients;
... populate set ...
set<name>::iterator pos;
pos = clients.find(name("Mary","Smith"));
pos->_last = "Adams";
```

okay ?

original binary tree

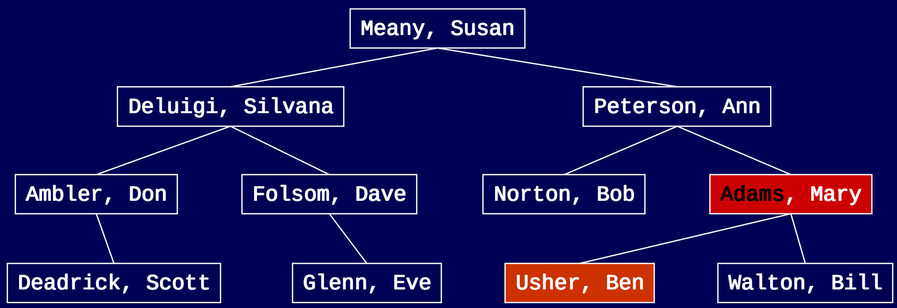



corrupted binary tree
Meany, Susan
Deluigi, Silvana
Peterson, Ann
Ambler, Don
Folsom, Dave
Norton, Bob
Adams, Mary
Deadrick, Scott
Glenn, Eve
Usher, Ben
Walton, Bill
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(9)


corrected binary tree
Meany, Susan
Deluigi, Silvana
Peterson, Ann
Ambler, Don
Folsom, Dave
Norton, Bob
Usher, Ben
Adams, Mary
Deadrick, Scott
Glenn, Eve
Walton, Bill
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(10)

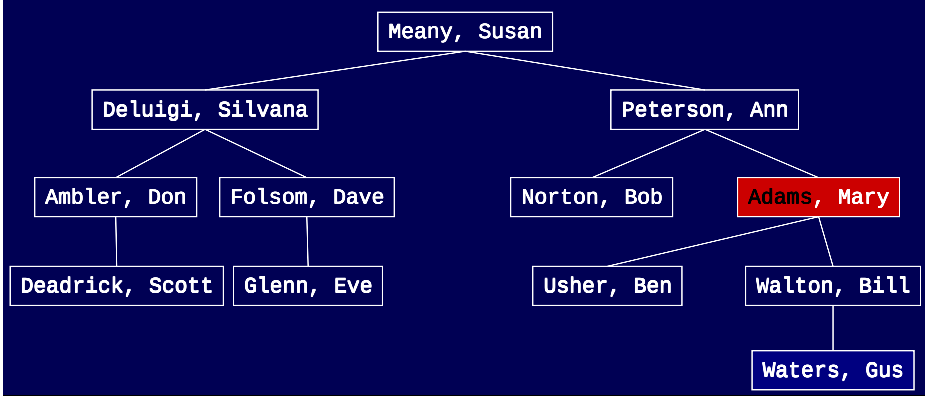
problems with corrupted tree

- can't find entries any longer
  - no Mary Adams
  - no Ben Usher



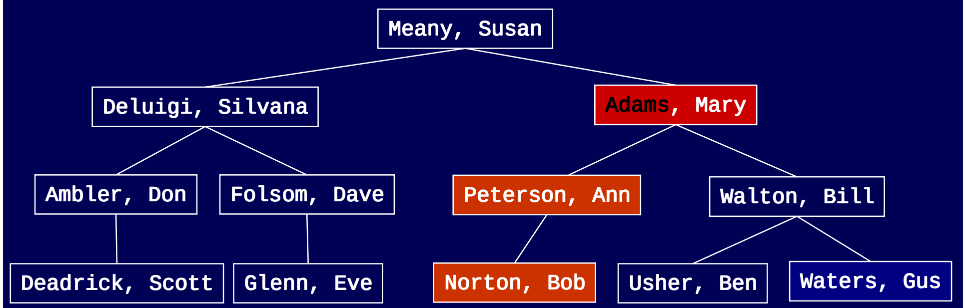
problems with corrupted tree

- insertion might make it worse
  - insert Gus Waters
  - tree is unbalanced



problems with corrupted tree

- re-balance tree
  - even more elements can’t be found



“modification” of set elements

- insert modified element and erase original
  - never modify an element in a `set`

```
set<string> clients;
... populate set ...

clients.insert(name(“Mary”,”Adams”));
clients.erase(name(“Mary”,”Smith”));
```

golden rule #1

- two means of access to elements in an STL container:
  - via container member functions
    - common: `erase(), insert()`
    - set-specific: `find(), count()`
  - via container iterators
    - `iterator, const_iterator, reverse_iterator`

Avoid modification of `set` elements through iterators; use member functions for modification of `set` elements.

agenda

- mechanics of tree-based containers
- dangerous algorithms
- read-only vs. read-write iterators
- iterator adaptors

a less obvious modification

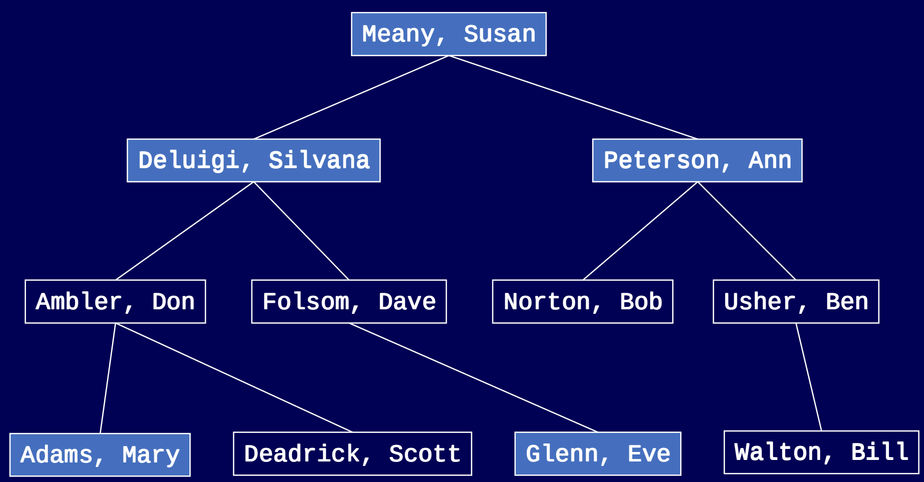
- algorithms use iterators
  - what is the consequence for use of algorithms in conjunction with the `set` container?
  - look into a couple of examples ...

- remove elements from `set`

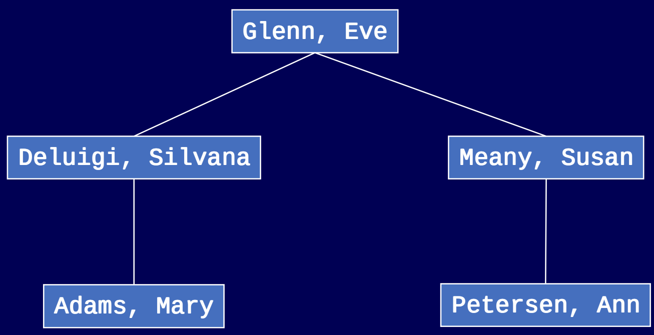
```
set<name> clients;
... populate set ...

remove_if(clients.begin(), clients.end(), isMale());
```


binary tree before remove_if()

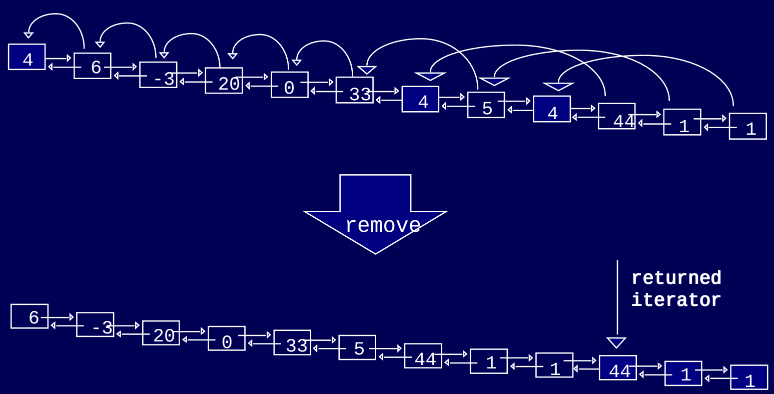


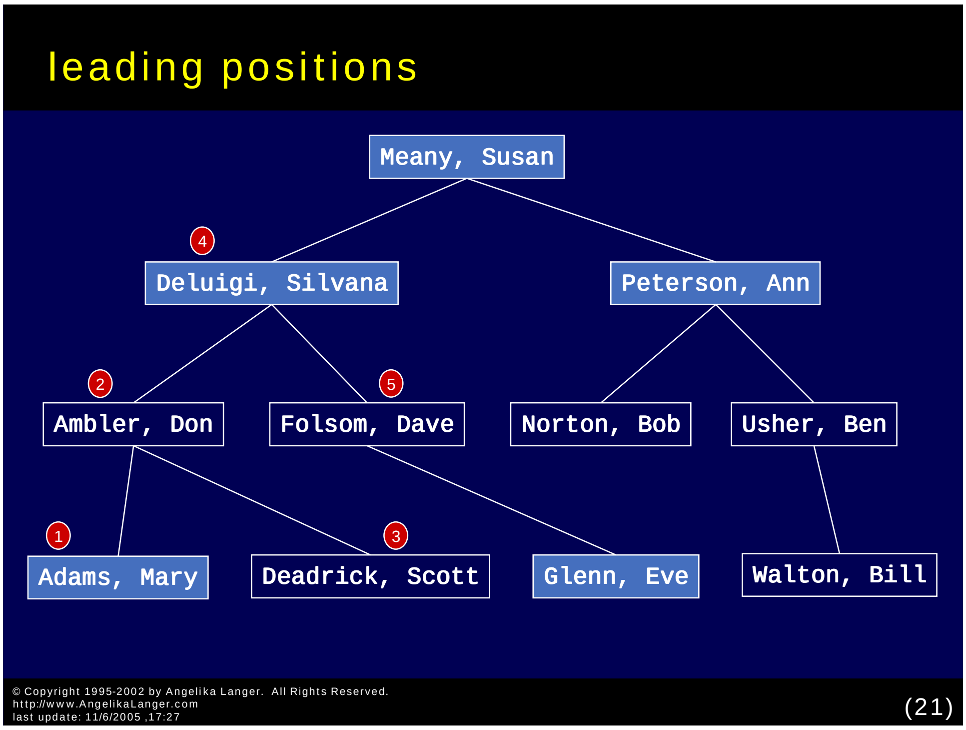
expected result

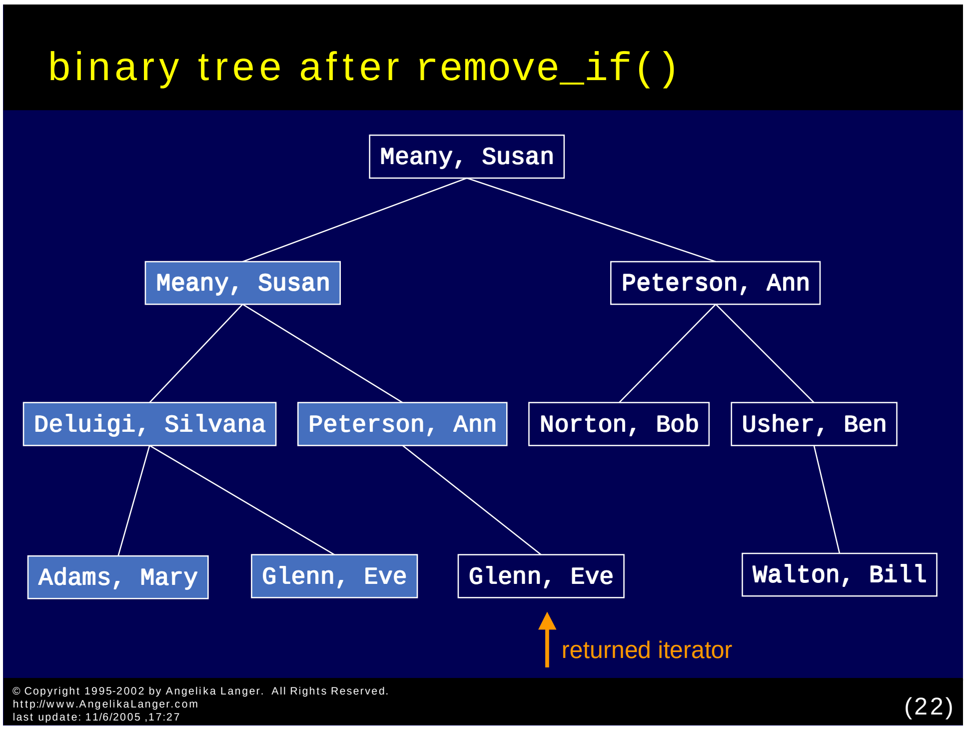


remove() algorithm

- `remove()` does not remove anything
  - copies valid elements to front and
  - returns iterator to garbage at end




leading positions
Meany, Susan
4
Deluigi, Silvana
Peterson, Ann
2
Ambler, Don
5
Folsom, Dave
Norton, Bob
Usher, Ben
1
Adams, Mary
3
Deadrick, Scott
Glenn, Eve
Walton, Bill
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(21)


binary tree after remove_if()
Meany, Susan
Meany, Susan
Peterson, Ann
Deluigi, Silvana
Peterson, Ann
Norton, Bob
Usher, Ben
Adams, Mary
Glenn, Eve
Glenn, Eve
Walton, Bill
returned iterator
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(22)

erase-remove

- erase garbage from `set`
  - not guaranteed to work because tree is corrupted

```
set<name> clients;
... populate set ...
clients.erase(
      remove_if(clients.begin(), clients.end(), isMale()),
      clients.end()
             );
```


what's the point?

- `remove()` is a mutating algorithm

- mutating algorithms
  - modify elements through dereferenced iterators
  - potentially corrupt the tree
  - cannot safely be applied to a tree-based sequence

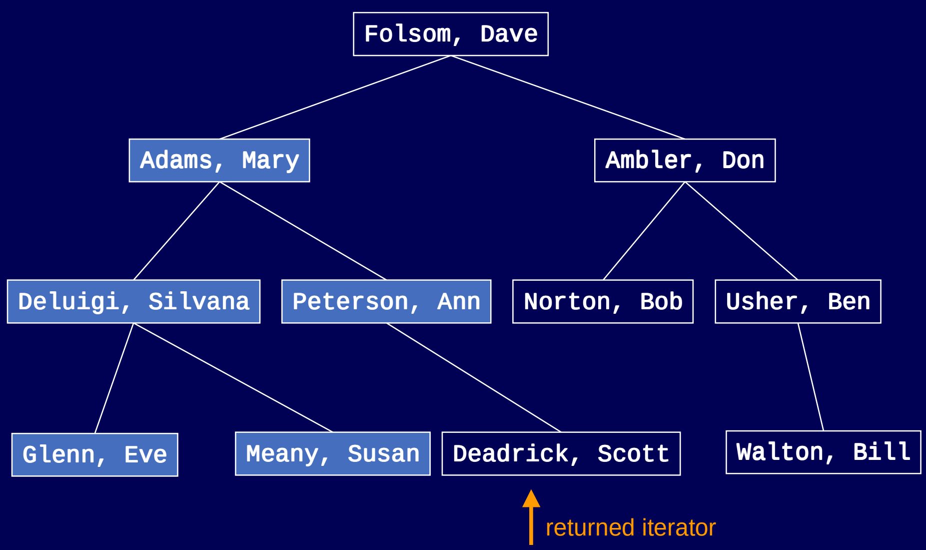
further pitfalls

- use `partition()` to find all females
  - `partition()` places all elements that satisfy a condition before all elements that do not satisfy it.

```
set<name> clients;
... populate set ...
set<name>::iterator res =
    partition(clients.begin(), clients.end(), isFemale());
```

- result:
  - `[clients.begin(), res)` is female
  - `[res, clients.end())` is male

after partition()



rule #2

Never apply a mutating algorithm to a tree-based sequence.

- tree-based containers in the STL

```
set         map
multiset    multimap
```


which are the mutating algorithms?

standard classifies algorithms into 4 categories:

- *non-modifying*
- *mutating*
- *sorting*
- *numeric*

confusing terminology:

- *mutating* algorithms need not be harmful
- *non-modifying* algorithms can be harmful
- *sorting* algorithms can be both harmless and harmful
- *numeric* algorithms are usually harmless

mutating algorithms

mutating algorithms modify

- either input sequence (in-place algorithm)
- or output sequence (copy algorithm)

example:

- `remove()` and `remove_copy()`
  - both listed as *mutating* algorithms
- `remove()` modifies the input sequence
  - harmful, can corrupt tree
- `remove_copy()` modifies only the output sequence
  - harmless for the input sequence
  - same for all algorithms that have an output range
    - `merge, transform, set_union, …`

golden rule #3

Never use a tree-based sequence as the output sequence of an algorithm.

non-modifying algorithms

non-modifying algorithms

– do not modify any sequence (neither input nor output)

example:

- `count_if()` and `for_each()`
  - listed as *non-mutating* algorithms
- can modify input sequence through function object
  - prohibited by the standard, but possible in practice
  - yet common with `for_each()`

sorting algorithms

sorting algorithms

– require sorted sequences

– some modify neither input nor output
  - harmless
  - example: `includes()`

– some modify only output
  - harmless
  - example: `merge(), set_union()`

– some modify input
  - dangerous
  - example: `inplace_merge(), sort()`

numeric algorithms

numeric algorithms

- reside in `<numeric>`, not `<algorithm>`

- `accumulate()` and `inner_product()`
  - produce numeric results
  - harmless
- `partial_sum()` and `adjacent_difference()`
  - modify an output sequence
  - harmless
- both take functors
  - must not have any side effects; required, but not enforced

functor pitfall

- count frequent flyers and raise their status

```
bool freqFlyer(clientRec& client)
{ if (client.getMiles() >= 1000000)
  { client.setStatus(GOLD); return true; }
  return false;
}
```

```
set<clientRec> clients;
... populate set ...
size_t cnt =
    count_if(clients.begin(), clients.end(), freqFlyer);
```

- clearly a modification of set elements
  - harmful if status contributes to sorting order
  - prohibited by the standard, but cannot be prevented

inside an algorithm

```
template <class InputIterator, class Predicate>
size_t count_if (InputIterator first, InputIterator last,
                ,Predicate pred)
{
  size_t cnt=0;
  while (first != end)
    if (pred(*first++)) ++cnt;
  return cnt;
}
```

- predicate can modify sequence element through dereferenced iterator
  - if argument is passed by reference

an alternative approach

- modification through functor of `for_each()`

```
class raiseStatus {
 size_t _cnt;
public:
 raiseStatus() : _cnt(0) { }
 void operator()(clientRec& client)
 { if (client.getMiles() >= 1000000)
   { client.setStatus(GOLD); ++_cnt; }
 }
 size_t getCnt() { return _cnt; }
};
```

```
set<clientRec> clients;
... populate set ...
size_t cnt =
    for_each(clients.begin(), clients.end(), raiseStatus())
    .getCnt();
```

golden rule #4

Functors must not modify sequence elements through the dereferenced iterator.

yet another approach

- modification through `transform()`

```
class raiseStatus {
 size_t* _cntPtr;
public:
 raiseStatus(size_t* p) : _cntPtr(p) { }
 clientRec operator()(clientRec client) const
 { if (client.getMiles() >= 1000000)
   { client.setStatus(GOLD); ++(*cnt); }
   return client;
 }
};
```

```
set<clientRec> clients;
... populate set ...
size_t cnt;
transform(clients.begin(), clients.end(), clients.begin(),
          raiseStatus(&cnt));
```

a typical transformation

- in-place transformation
  - output sequence is input sequence

```
clientRec raiseStatus(clientRec client)
{ if (client.getMiles() >= 1000000)
  {  client.setStatus(GOLD);  }
  return c;
}
```

```
set<clientRec> clients;
... populate set ...
size_t cnt =
    transform(clients.begin(), clients.end(),
              clients.begin(), raiseStatus());
```


golden rules for algorithms and set

#2 Never use a tree-based sequence as the output sequence of any algorithm.

#3 Never use functors that modify sequence elements through the dereferenced iterator.

#4 Never use a tree-based sequence as input sequence of a mutating algorithm that modifies the input sequence.

“dangerous” algorithms

- algorithms that modify an output sequence
  - ☑ golden rule #2

copy copy_backward replace_copy replace_copy_if merge	remove_copy remove_copy_if unique_copy reverse_copy rotate_copy swap swap_ranges	set_union set_intersection set_difference set_symmetric_difference partial_sort_copy transform

“dangerous” algorithms

- algorithms that take predicates (or other functors)
  - ☑ golden rule #3

find_if find_end find_first_of adjacent_find search search_n count_if mismatch equal	replace_if replace_copy_if remove_if remove_copy_if unique unique_copy partition stable_partition	transform for_each

“dangerous” algorithms

- algorithms that take comparators
  - ☑ golden rule #3

```
sort
stable_sort
partial_sort
partial_sort_copy
nth_element

lower_bound
upper_bound
equal_range
binary_search

next_permutation
previous
    _permutation

min
max
min_element
max_element
lexicographical
    _compare

merge
inplace_merge

includes
set_union
set_intersection
set_difference
set_symmetric
    _difference
```


“dangerous” algorithms

- algorithms that actively modify the input sequence
  - ☑ golden rule #4

```
replace
replace_if

fill
fill_n
generate
generate_n

remove
remove_if
unique

inplace_merge

reverse
rotate

swap
swap_ranges

random_shuffle
next_permutation
previous_permutation

partition
stable_partition

sort
stable_sort
partial_sort
nth_element
```

agenda

- mechanics of tree-based containers
- dangerous algorithms
- read-only vs. read-write iterators
- iterator adapters

solution

- goal: prevent inadvertent corruption of tree
- STL implementations if `set` take different approaches

[1] regular read-write iterators

» modification is possible
» requires programming discipline; stick to the rules
» few implementations, e.g. MVC 6.0

[2] read-only iterators

» `iterator` type is same as `const_iterator`
» restrictive; no modification possible at all
» many implementations, e.g. SGI, Metroworks

read-only set iterators

- safe side of the coin
  - catches all attempts to modify elements in the tree through iterators
  - i.e. catches all violations of rules #1-#4
- restrictive side of the coin
  - often not all parts of an element contribute to the sorting order
  - these parts could safely be modified
  - read-only iterators prevent even harmless modifications

case study: set of bank accounts

- set of bank accounts
  - bank account class is legacy code; cannot be changed
  - only account # determines sorting order

```
class account {
 size_t _number;   // determines ordering
 double _balance; // irrelevant for ordering
 …
};
bool operator<(const account& lhs, const account& rhs)
{ return lhs._number < rhs._number; }
```

attempted modification

- blatant attempt to destroy the tree
  - replace entire element including account number
  - rightly rejected

```
set<account> clients;
…
set<account>::iterator iter;
…
*iter = *new account;  // error: modification of key!
```


reasonable modification

- harmless modification
  - balance does not contribute to sorting order
  - rejected - what can we do?

```
set<account> clients;
…
set<account>::iterator iter;
…
iter->_balance = 1000000; // harmless: does not affect key!
```

solution by brute force

- cast away constness

```
const_cast<double&>(iter->_balance) = 1000000;
```

how does it work?

- `set<account>::iterator` is a `const_interator`
- `*iter` yields reference of type `const account&`
- `iter->balance` is a `const double&`
- cast away the reference's constness

note:

- `const_cast` only allowed on references and pointers

a more sophisticated approach

- find a portable solution
  - hide away the implementation differences
  - encapsulate `const_cast` somehow
- idea:
  - add `const` member function `setBalance()` to `account` class
  - bad idea:
    - semantically wrong
    - `setBalance()` is not an inspecting function
    - would allow modification even on `const account` objects
- a better idea:
  - solve problem where it arises
    - change iterator type
    - build iterator adapter

iterator adapter

- iterator adapter `balanceIter`
  - adapts the set iterator
  - gives write-access to part that can safely be modified
  - no access to critical parts such as account number
- special dereference operator
  - returns a non-const reference to balance of element pointed to

instead of

```
iter->_balance = 1000000;
```

use

```
*balanceIter(iter) = 1000000;
```


sketch of an implementation

```
class balanceIter {
public:
 explicit balanceIter(set<account>::iterator i) :_i(i) {}
 double& operator*() const
 { return const_cast<double&>(_i->_balance); }
 balanceIter& operator++() { ++_i; return *this; }
 // ... postfix ++, pre- and postfix -- ...
private:
 set<account>::iterator _i;
};
```

- principles:
  - built on top of original set iterator
  - adaptation happens in `operator*`
  - remaining iterator operations are simple delegations

advantages of iterator adapter

- easy to port to a different STL implementation
  - `const_cast` hidden in `operator*`
  - need not even remove the `const_cast`
    - cast simply not needed in implementations with read-write set iterators
- adapted iterator can be supplied to algorithms
  - can safely relax the golden rules
  - can use algorithms to perform modification on mutable parts of the elements

without iterator adapter

- add interest to balance on all accounts

```
void addInterest(account& acc) { acc._balance =* 1.025; }
```

```
set<account> clients;
…
for_each(clients.begin(), clients.end(), addInterest);
```

does not compile

- inside `for_each`:
  - dereferenced iterator yields `const` reference to `account`

```
template <class InputIterator, class Functor>
Functor for_each(InputIterator first, InputIterator last,
                ,Functor fct)
{ while (first != end) fct(*first++); return fct; }
```

with iterator adapter

- could solve problem by `const_cast` in functor
- use iterator adapter instead
  - hides away the platform difference
  - adapted iterator yields non-`const` reference to `account.balance`

```
void addInterest(double& bal) { bal =* 1.025; }
```

```
set<account> clients;
…
for_each(balanceIter(clients.begin()),
         balanceIter(clients.end()),
         addInterest);
```

works

- gain through adapter:
  - need not be aware in all places of the platform differences

adapter enables code reuse

- it is more likely that you already have a functor like this:

```
class interestAdder {
   const double _rate;
public:
   interestAdder(double r) : _rate(1+(r/100.0)) {}
   double operator()(double bal) { return bal * _rate; }
}
```

- rather than a functor like this:

```
class interestAdder {
   const double _rate;
public:
   interestAdder(double r) : _rate(1+(r/100.0)) {}
   account operator()(account acc)
   { acc._balance * _rate; return acc; }
}
```

without adapter

```
class interestAdder {
   const double _rate;
public:
   interestAdder(double r) : _rate(1+(r/100.0)) {}
   account operator()(account acc)
   { return acc._balance * _rate; }
}
```

```
set<account> clients;
…
transform(clients.begin(), clients.end(), clients.begin(),
          interestAdder(2.5)
         );
```


with adapter

```
class interestAdder {
   const double _rate;
public:
   interestAdder(double r) : _rate(1+(r/100.0)) {}
   double operator()(double bal) { return bal * _rate; }
}
```

```
set<account> clients;
…
transform(balanceIter(clients.begin()),
          balanceIter(clients.end()),
          balanceIter(clients.begin()),
          interestAdder(2.5)
         );
```

- gain through adapter: reuse of existing functions

another advantage of adapter

- without adapter: need functor to calculate sum of balances

```
double total = accumulate(clients.begin(), clients.end(),
                          0.0, balanceAddition);
```

```
double balanceAddition(const account& a1, const account& a2)
{ return a1._balance + a2._balance; }
```

- adapter eases use of algorithms

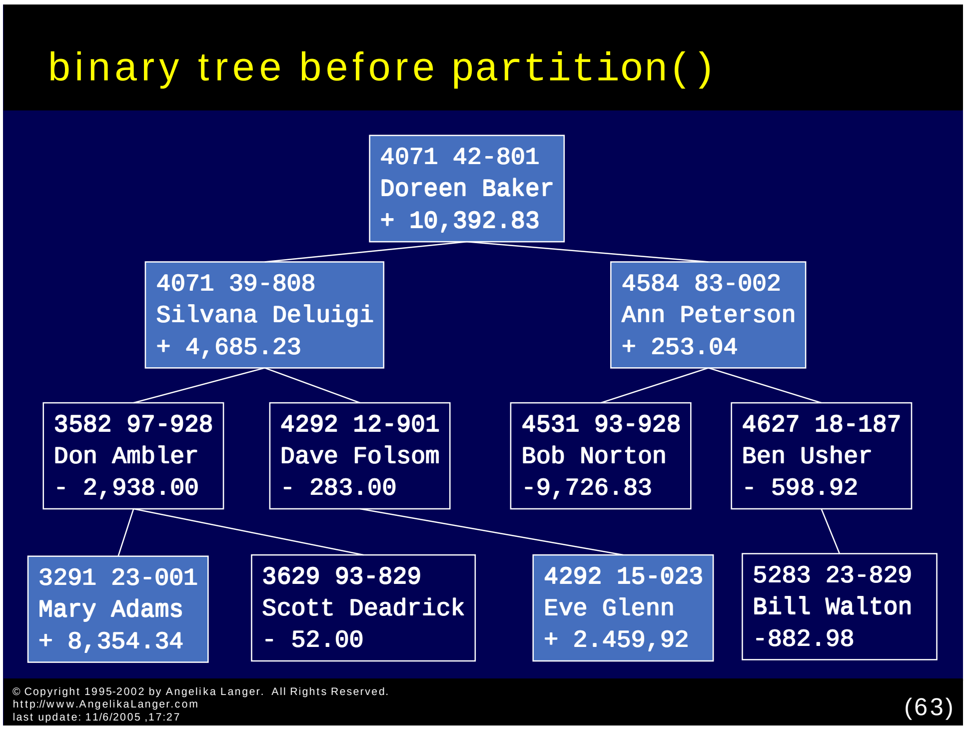
```
double total = accumulate(balanceIter(clients.begin()),
                          balanceIter(clients.end()),
                          0.0);
```

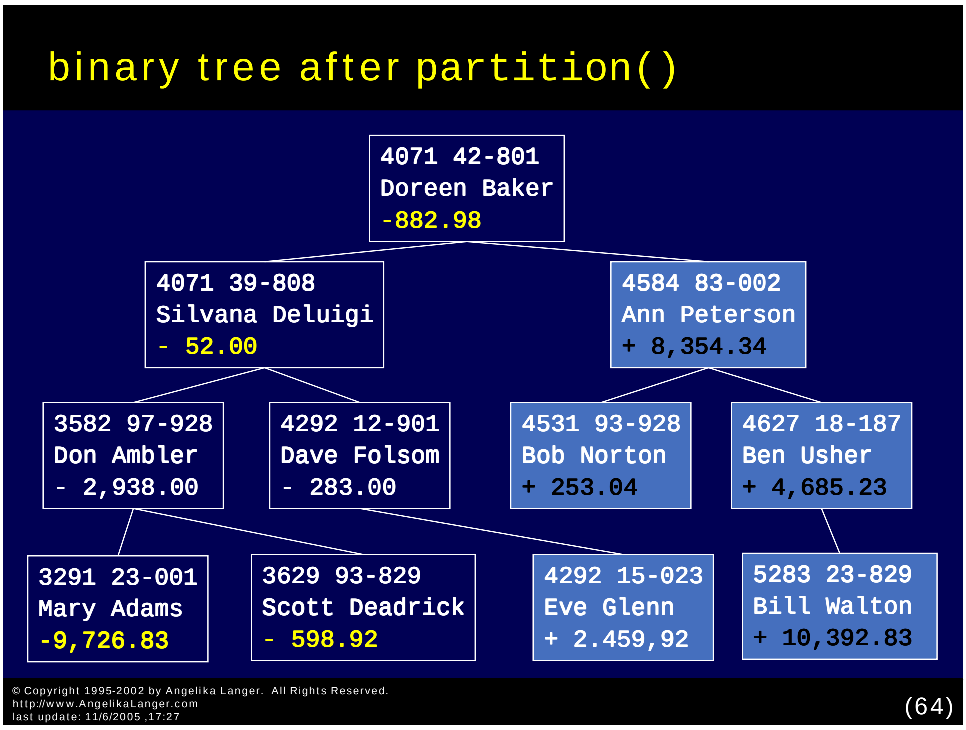

peril of adapter

- can corrupt collection
  - cannot corrupt the sorting order
  - but can produce inconsistent elements

```
bool inDebt(double d) { return d < 0.0; }
```

```
set<account>::iterator pos
    = partition(balanceIter(clients.begin()),
                balanceIter(clients.end()),
                inDebt);
```


binary tree before partition()
4071 42-801
Doreen Baker
+ 10,392.83
4071 39-808
Silvana Deluigi
+ 4,685.23
4584 83-002
Ann Peterson
+ 253.04
3582 97-928
Don Ambler
- 2,938.00
4292 12-901
Dave Folsom
- 283.00
4531 93-928
Bob Norton
-9,726.83
4627 18-187
Ben Usher
- 598.92
3291 23-001
Mary Adams
+ 8,354.34
3629 93-829
Scott Deadrick
- 52.00
4292 15-023
Eve Glenn
+ 2.459,92
5283 23-829
Bill Walton
-882.98
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(63)


binary tree after partition()
4071 42-801
Doreen Baker
-882.98
4071 39-808
Silvana Deluigi
- 52.00
4584 83-002
Ann Peterson
+ 8,354.34
3582 97-928
Don Ambler
- 2,938.00
4292 12-901
Dave Folsom
- 283.00
4531 93-928
Bob Norton
+ 253.04
4627 18-187
Ben Usher
+ 4,685.23
3291 23-001
Mary Adams
-9,726.83
3629 93-829
Scott Deadrick
- 598.92
4292 15-023
Eve Glenn
+ 2.459,92
5283 23-829
Bill Walton
+ 10,392.83
© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005 ,17:27
(64)

reality check

- use original read-only iterator will fail

```
bool inDebt(const account& a) { return a._balance < 0.0; }
```

```
set<account>::iterator pos
 = partition(clients.begin(), clients.end(), inDebt);
```

does not compile

- conclusion:
  - use of `partition()` on `set` not sensible
  - abuse through adapter cannot be prevented

- still need to avoid the “dangerous” algorithms
  - use of `set` as input to modifying algorithms still lethal (rule # 4)

evaluation of adapter

- iterator adapter - advantages
  - facilitates portability
  - enables use of existing pieces of code
  - permits use of set with mutating algorithms (relaxes rule #2)
  - permits use of set as output sequence (relaxes rule #3)
  - permits use of modifying functors (relaxes rule #4)

- iterator adapter - downsides
  - effort to implement the adapter
  - not fool-proof; can be abused
    - modifications must be sensible
    - cannot corrupt the tree, but lead to surprise

a word on map and hash_set

same situation for `hash_set`

- element modification through iterators must be prevented
  - content of element determines hash value
  - hash value determines position in data structure (index of bucket)
  - direct modification of element corrupts internal structure

different situation for `map`

- sorting order is protected via constness of key
  - `map` contains `pair<const Key, Value>`
- no need for a read-only map iterator

agenda

- mechanics of tree-based containers
- dangerous algorithms
- read-only vs. read-write set iterators
- iterator adapters

user-defined iterator types

an iterator type must provide:

- a number of nested types (`iterator_category`, `value_type`, etc.)
- copy constructor, copy assignment, and destructor
- equality comparisons `operator==()` and `operator!=()`
- dereference operators `operator*()` and `operator->()`
- prefix and postfix increment (and decrement)
- pointer arithmetics and comparison (random access)

required nested types

- needed to make iterator type adaptable

`iterator_category`

iterator concept that the iterator implements

input, output, forward, bidirectional, random access

`value_type`

type of element that the iterator points to

`difference_type`

type to express the distance between two iterators

`pointer`

pointer type to an element; returned by `operator->`

`reference`

reference type to an element; returned by `operator*`

user-defined iterator adapter types

iterator adapter types

- contain the adapted iterator as a data member,
- implement their functionality in terms of the underlying iterator, and
- have a `base()` member function that yields the underlying iterator

implementation of adapter

```
class balanceIter {
public:
 // constructors
 balanceIter() {}
 explicit balanceIter(set<account>::iterator i) :_i(i) {}

 // conversion back to underlying type
 set<account>::iterator base() const { return _i; }

private:
 set<account>::iterator _i;
};
```

implementation of adapter

```
class balanceIter {
public:
 // required nested types
 typedef set<account>::iterator setIterator;
 typedef setIterator::iterator_category iterator_category;
 typedef double value_type;
 typedef setIterator::difference_type difference_type;
 typedef double pointer;
 typedef double reference;
};
```

- nested types:
  - same as for underlying set iterator

implementation of adapter

```
class balanceIter {
public:
 // derefence operators
 double& operator*() const
 { return const_cast<double&>(_i->_balance); }
 double* operator->() const
 { return const_cast<double*>(&_i->_balance); }

};
```

- actual adaptation:
  - return non-const reference and non-const pointer to balance

implementation of adapter

```
class balanceIter {
public:
 // increment / decrement operators
  balanceIter& operator++()
  { ++_i; return *this; }
  balanceIter operator++(int)
  { balanceIter tmp = *this;
    ++_i; return tmp;
  }
  ... same for decrement ...
};
```

- increment / decrement do not change:
  - simple delegation to underlying set iterator

implementation of adapter

```
bool operator==(const balanceIter& x, const balanceIter& y)
{ return x.base() == y.base(); }

bool operator!=(const balanceIter& x, const balanceIter& y)
{ return !(x==y); }
```

- comparison does not change:
  - simple delegation to underlying set iterator

refinements

- might want to allow conversions between adapter types

```
class balanceIter {
public:
 // constructors
 explicit balanceIter(nameIter i)    :_i(i.base()) {}
 explicit balanceIter(addressIter i) :_i(i.base()) {}
 ...
private:
 set<account>::iterator _i;
};
```


conversions between adapter types

- use conversions between adapter types

```
// search for name
nameIter pos =
  find(nameIter(clients.begin()), nameIter(clients.end()),
       name("Eve","Glenn"));
// change address
*addressIter(pos) = address("736 12th St.",
                            "Albany, TX 97263"
                            "USA"
                           );
```

- otherwise:

```
(pos.base())->_address = address(...);
```

wrap-up

- tree-based containers need to preserve their internal structure
  – undefined behavior if tree is corrupted
- modifications through iterators are potentially dangerous
  – can happen inadvertently through use of algorithms or mutating functors
- STL implementations differ in how they address the problem
  – read-only vs. read-write iterators for `set` and `hash_set`
  – constant key for `map`
- iterator adapters hide away the differences
  – facilitate code reuse
  – simplify use of algorithms and implementation of functors
  – are relatively easy to implement and use

contact info

Angelika Langer

Training & Mentoring

Object-Oriented Software Development in C++ & Java

email: info@AngelikaLanger.com

http: //www.AngelikaLanger.com