Coping with Read-Only Set Iterators

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

Meany, Alvin

Deluigi, Silvana

Ambler, Don

Deadrick, Scott

Folsom, Dave

Glenn, Eve

Norton, Bob

Usher, Ben

Peterson, Ann

Smith, Mary

Walton, Bill
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";

original binary tree

Meany, Susan

Deluigi, Silvana

Ambler, Don

Folsom, Dave

Norton, Bob

Smith, Mary

Deadrick, Scott

Glenn, Eve

Usher, Ben

Walton, Bill
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

```
     Meany, Susan
    /       \
  Deluigi, Silvana         Adams, Mary
   /       \
Ambler, Don  Folsom, Dave
   /       \
Deadrick, Scott        Peterson, Ann
     /       \    
    Glenn, Eve  Norton, Bob
          /       \
        Usher, Ben  Waters, Gus
```

“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`

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

binary tree before `remove_if()`

- Meany, Susan
- Deluigi, Silvana
- Peterson, Ann
- Ambler, Don
- Folsom, Dave
- Norton, Bob
- Usher, Ben
- Adams, Mary
- Deadrick, Scott
- Glenn, Eve
- Walton, Bill
expected result

Glenn, Eve

Deluigi, Silvana

Adams, Mary

Meany, Susan

Petersen, Ann

remove() algorithm

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

! remove() does not remove anything
– copies valid elements to front and
– returns iterator to garbage at end
leading positions

Meany, Susan

Deluigi, Silvana

Peterson, Ann

Ambler, Don
Folsom, Dave
Norton, Bob
Usher, Ben

Adams, Mary
Deadrick, Scott
Glenn, Eve
Walton, Bill

binary tree after remove_if()
erase-‐remove

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

```cpp
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.

```cpp
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()`**

- `Folsom, Dave`
- `Adams, Mary`
- `Ambler, Don`
- `Deluigi, Silvana`
- `Peterson, Ann`
- `Norton, Bob`
- `Usher, Ben`
- `Glenn, Eve`
- `Meany, Susan`
- `Deadrick, Scott`
- `Walton, Bill`

Returned iterator
**rule #2**

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

- tree-based containers in the STL
  - `set`
  - `multiset`
  - `map`
  - `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**

- 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

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

```cpp
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 getCount() { return _cnt; }
  };

  set<clientRec> clients;
  ... populate set ...
  size_t cnt =
    for_each(clients.begin(), clients.end(), raiseStatus()).getCount();
golden rule #4

Functors must not modify sequence elements through the dereferenced iterator.

yet another approach

- modification through \texttt{transform()}

```cpp
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(), clients.begin(), raiseStatus(&cnt));
```
**a typical transformation**

- in-place transformation
  - output sequence is input sequence

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

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

---

**golden rules for algorithms and set**

1. Never use a tree-based sequence as the output sequence of any algorithm.
2. Never use functors that modify sequence elements through the dereferenced iterator.
3. Never use a tree-based sequence as input sequence of a mutating algorithm that modifies the input sequence.
### “dangerous” algorithms

1. **algorithms that modify an output sequence**
   - **golden rule #2**

<table>
<thead>
<tr>
<th>Modify Output Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy</code></td>
</tr>
<tr>
<td><code>copy_backward</code></td>
</tr>
<tr>
<td><code>replace_copy</code></td>
</tr>
<tr>
<td><code>replace_copy_if</code></td>
</tr>
<tr>
<td><code>merge</code></td>
</tr>
<tr>
<td><code>remove_copy</code></td>
</tr>
<tr>
<td><code>remove_copy_if</code></td>
</tr>
<tr>
<td><code>unique_copy</code></td>
</tr>
<tr>
<td><code>reverse_copy</code></td>
</tr>
<tr>
<td><code>rotate_copy</code></td>
</tr>
<tr>
<td><code>swap</code></td>
</tr>
<tr>
<td><code>swap_ranges</code></td>
</tr>
<tr>
<td><code>set_union</code></td>
</tr>
<tr>
<td><code>set_intersection</code></td>
</tr>
<tr>
<td><code>set_difference</code></td>
</tr>
<tr>
<td><code>set_symmetric_difference</code></td>
</tr>
<tr>
<td><code>partial_sort_copy</code></td>
</tr>
<tr>
<td><code>transform</code></td>
</tr>
</tbody>
</table>

2. **algorithms that take predicates (or other functors)**
   - **golden rule #3**

<table>
<thead>
<tr>
<th>Find/Match/Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find_if</code></td>
</tr>
<tr>
<td><code>find_end</code></td>
</tr>
<tr>
<td><code>find_first_of</code></td>
</tr>
<tr>
<td><code>adjacent_find</code></td>
</tr>
<tr>
<td><code>search</code></td>
</tr>
<tr>
<td><code>search_n</code></td>
</tr>
<tr>
<td><code>count_if</code></td>
</tr>
<tr>
<td><code>mismatch</code></td>
</tr>
<tr>
<td><code>equal</code></td>
</tr>
<tr>
<td><code>replace_if</code></td>
</tr>
<tr>
<td><code>replace_copy_if</code></td>
</tr>
<tr>
<td><code>remove_if</code></td>
</tr>
<tr>
<td><code>remove_copy_if</code></td>
</tr>
<tr>
<td><code>unique</code></td>
</tr>
<tr>
<td><code>unique_copy</code></td>
</tr>
<tr>
<td><code>partition</code></td>
</tr>
<tr>
<td><code>stable_partition</code></td>
</tr>
<tr>
<td><code>transform</code></td>
</tr>
<tr>
<td><code>for_each</code></td>
</tr>
</tbody>
</table>
### “dangerous” algorithms

- **algorithms that take comparators**
  - golden rule #3

<table>
<thead>
<tr>
<th>sort</th>
<th>next_permutation</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>stable_sort</td>
<td>previous_permutation</td>
<td>inplace_merge</td>
</tr>
<tr>
<td>partial_sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>partial_sort_copy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nth_element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lower_bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>upper_bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equal_range</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary_search</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **golden rule #3**

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

<table>
<thead>
<tr>
<th>replace</th>
<th>inplace_merge</th>
<th>partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace_if</td>
<td></td>
<td>stable_partition</td>
</tr>
<tr>
<td>fill</td>
<td>reverse</td>
<td>sort</td>
</tr>
<tr>
<td>fill_n</td>
<td>rotate</td>
<td>stable_sort</td>
</tr>
<tr>
<td>generate</td>
<td>swap</td>
<td>partial_sort</td>
</tr>
<tr>
<td>generate_n</td>
<td>swap_ranges</td>
<td>n nth_element</td>
</tr>
<tr>
<td>remove</td>
<td>random_shuffle</td>
<td>next_permutation</td>
</tr>
<tr>
<td>remove_if</td>
<td>previous_permutation</td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```cpp
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

```cpp
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?

```cpp
set<account> clients;
...
set<account>::iterator iter;
...
iter->_balance = 1000000; // harmless: does not affect key!
```
solution by brute force

- cast away constness

\[
\text{const_cast<double&>(iter->_balance) = 1000000;}
\]

how does it work?
- \text{set<account>::iterator is a const_iterator}
- \*iter yields reference of type const account&
- \*iter->balance is a const double&
- cast away the reference’s constness

note:
- \text{const_cast} only allowed on references and pointers

a more sophisticated approach

- find a portable solution
  - hide away the implementation differences
  - encapsulate \text{const_cast} somehow

idea:
- add \text{const} member function \text{setBalance()} to \text{account} class
  - bad idea:
    - semantically wrong
    - \text{setBalance()} is not an inspecting function
    - would allow modification even on \text{const} \text{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**

```cpp
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

```cpp
does not compile

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

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

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

```cpp
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-constant reference to `account.balance`

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

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

- 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:

```cpp
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:

```cpp
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

```cpp
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

```cpp
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

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

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

- adapter eases use of algorithms

```cpp
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

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

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

- use original read-only iterator will fail

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

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

- 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

```cpp
does not compile```


**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 \( \text{operator}==() \) and \( \text{operator}!==(\) 
- dereference operators \( \text{operator}*(\) and \( \text{operator}->() \)
- 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
  - 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 \( \text{operator}->() \)
  - reference
    - reference type to an element; returned by \( \text{operator}*(\)
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 \texttt{base()} member function that yields the underlying iterator

implementation of adapter

\begin{verbatim}
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;
};
\end{verbatim}
implementation of adapter

```cpp
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

```cpp
class balanceIter {
public:
    // dereference 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

```cpp
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

```cpp
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

```cpp
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

```cpp
// 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:

```cpp
(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