# STL Gotchas

## Avoiding Common Errors in Using the STL

**Angelika Langer**

Trainer/Consultant

`http://www.camelot.de/~langer/`

# Agenda

- Insertion and deletion might invalidate references, pointers, and iterators.
- A set iterator must not allow modification of the elements.
- Function objects must neither have side effects nor modify container elements.
- Comparators must not be polymorphic.
- Associative containers use induced equivalence, whereas algorithms use equality.
- Adapted iterators cannot be passed to container operations.
- Stream iterators on the same stream are not independent of each other.
- Allocators must exhibit static behavior.

# STL Pitfall #1

Invalidation of
references, pointers, and
iterators

# All container in the STL ...

- are of dynamic size, i.e. they grow as needed
- allow insertion and removal of elements via member function `insert()` and `erase()`
- provide iterators that give access to the contained elements
- provide iterators to the beginning and end of the sequence

Consider a program that
 reads lines from an input file,
 sorts them, and
 writes the result of sorting to an output file,
 using a `list` as the temporary store.

# Inserting elements to a list

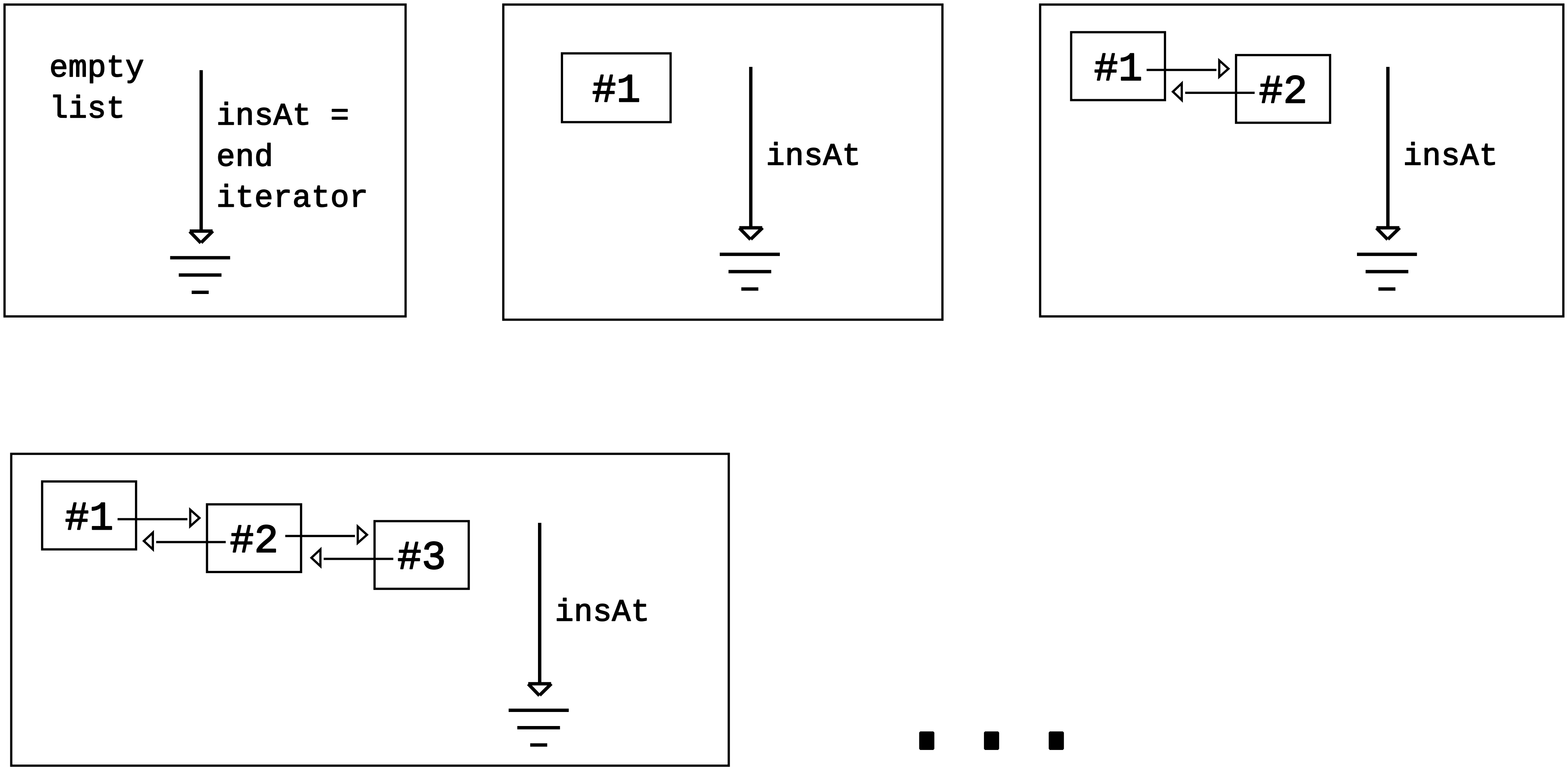
```
void doIt(const char* in,const char* out)
{  list<string> buf;
   list<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
          buf.insert(insAt,linBuf);

   buf.sort();

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),
        ostream_iterator<string>(outFile,"\n"));
}
```

# Insert at a specified position

```
iterator list::insert(iterator position,
                      const value_type& value)
```

- Inserts a copy of `value` before the specified `position`.
- The returned iterator points to the newly inserted copy of `value`.

# Inserting elements to a list



# Inserting elements to a set

```
void doIt(const char* in,const char* out)
{  set<string> buf;
   set<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
          buf.insert(insAt,linBuf);

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),
        ostream_iterator<string>(outFile,"\n"));
}
```

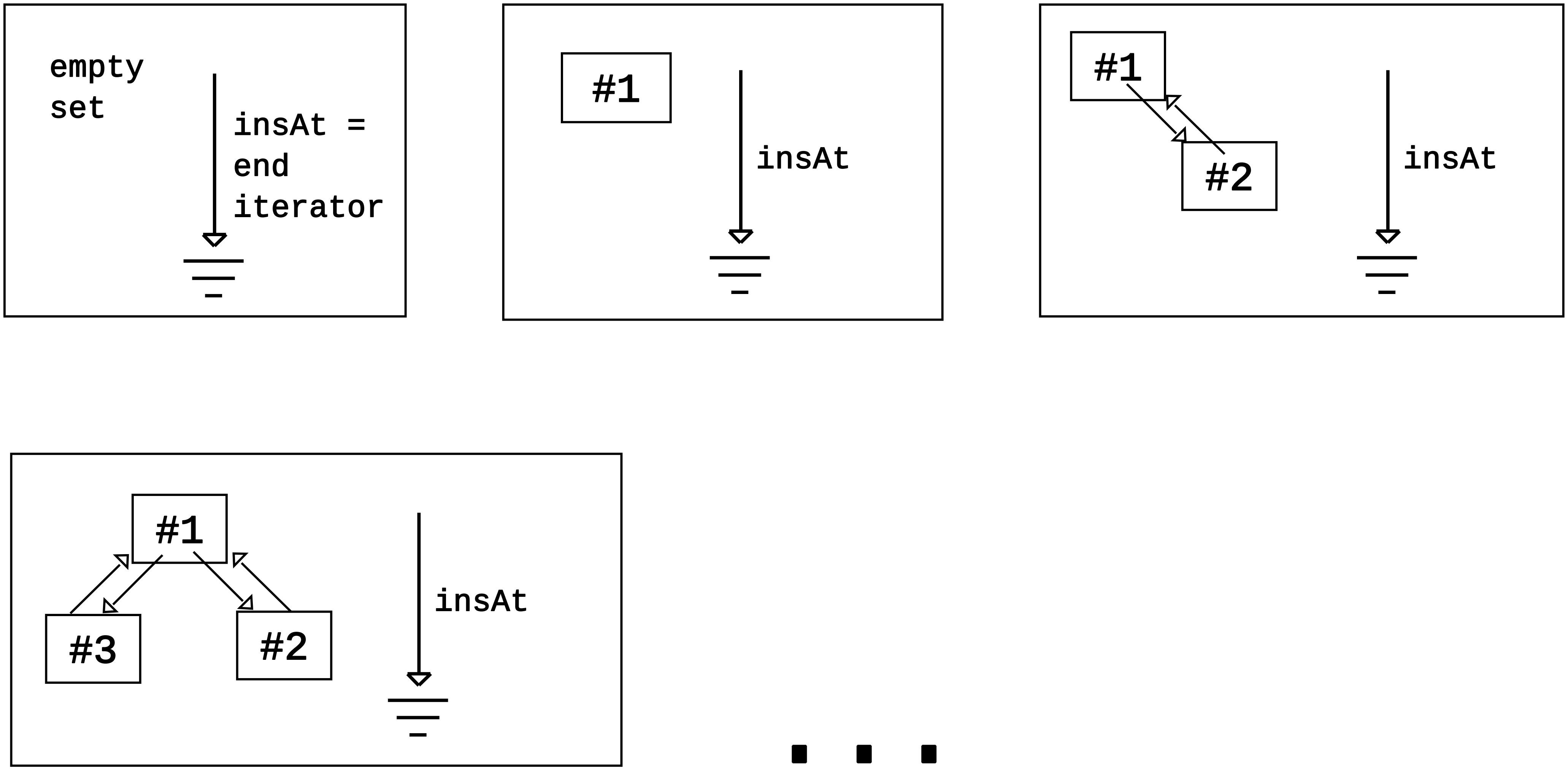
# Insert with hint

```
iterator set::insert(iterator position,
                     const value_type& value)
```

- Inserts a copy of `value`.
- Inserts only if there is no element in the container with the same value.
- The returned iterator points to the element with the same value.

- The iterator `position` is a hint. It should point to where the value should be inserted.
- The hint has no effect on correctness, only on performance.

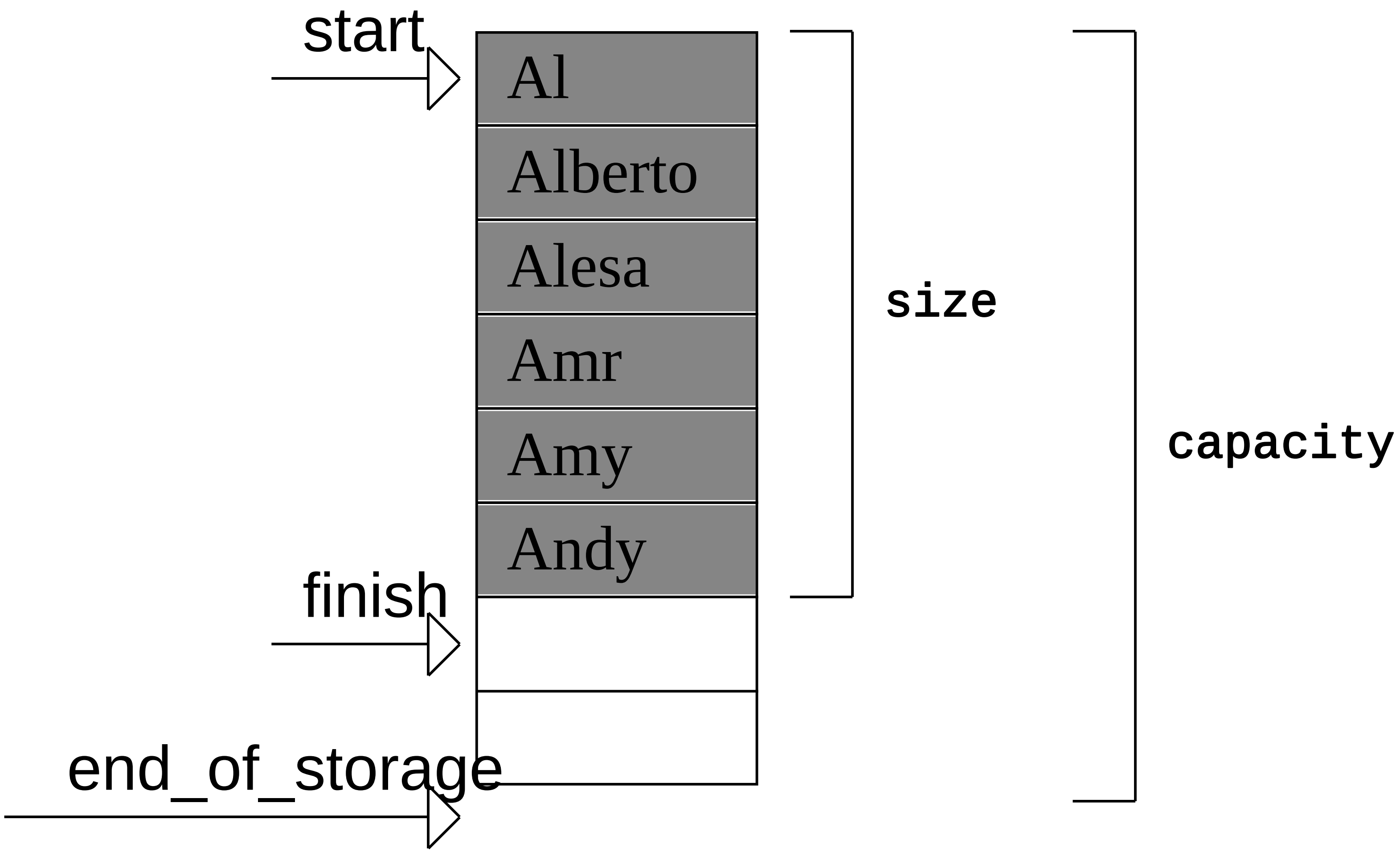
# Inserting elements to a set



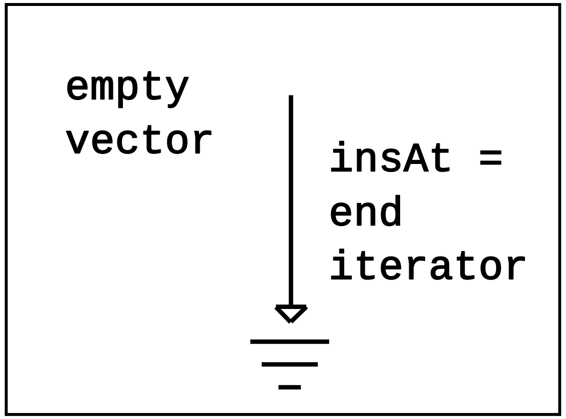
# Inserting elements to a vector

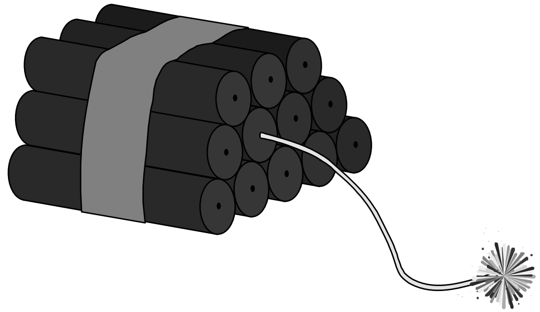
```
void doIt(const char* in,const char* out)
{  vector<string> buf;
   vector<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.insert(insAt,linBuf);   <<<<<< crash !!!

   sort(buf.begin(),buf.end());

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),
        ostream_iterator<string>(outFile,"\n"));
}
```

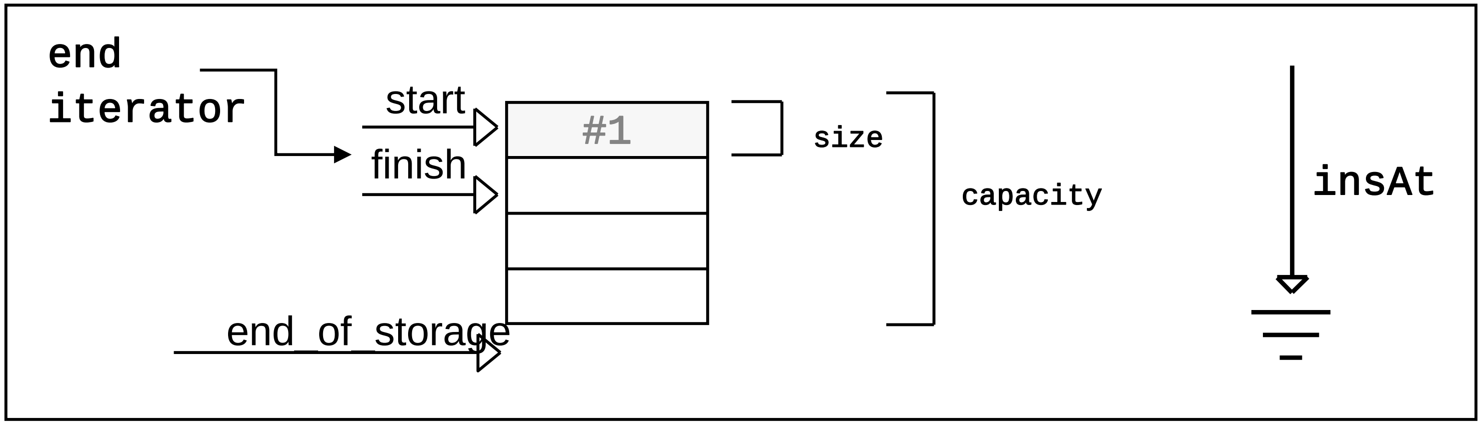
# Typical implementation of vector



# Inserting elements to a vector





# So, what's going wrong ... ?

- `doIt()` is built on the assumption that an iterator (`insAt = buf.end()`), that is valid in one context (with an empty vector), is still valid in another context (after insertion to the vector).
- There is no such guarantee.

```
void doIt(const char* in,const char* out)
{  ...
   vector<string>::iterator insAt = buf.end();
   ...
   while(getline(inFile,linBuf))
         buf.insert(insAt,linBuf);
   ...
}
```

# Validity guarantees for insertion

For associative containers and list:

- Does not affect the validity of iterators and references.

For vector:

- Causes reallocation if the new size is greater than the old capacity.
- Reallocation invalidates *all* the references, pointers, and iterators.
- If no reallocation happens, all the iterators and references before the insertion point remain valid.

# Inserting elements to a vector

Does not only apply to the end iterator, but to *any* iterator after the point of insertion:

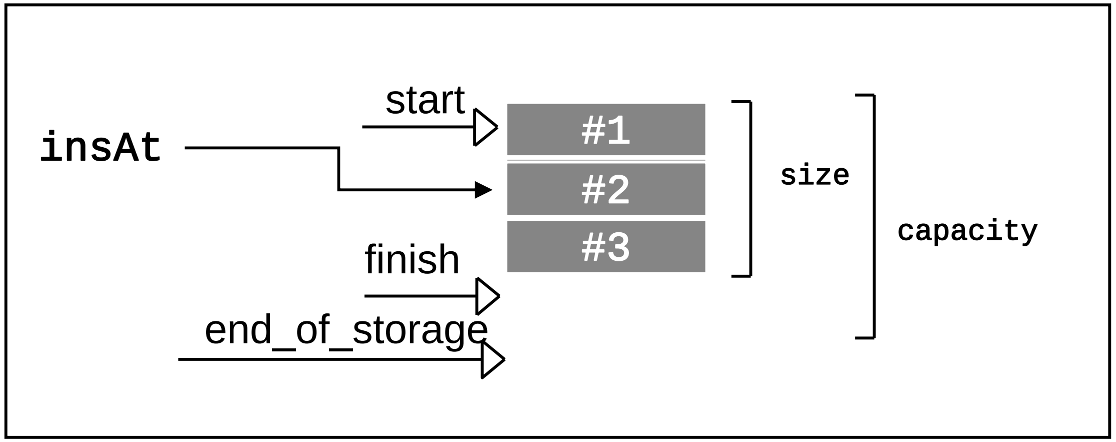
- Insertion moves all elements after the point of insertion to the back.
- All references to elements after the point of insertion become invalid.
- In particular, the point of insertion itself becomes invalid as a side effect of the insertion.

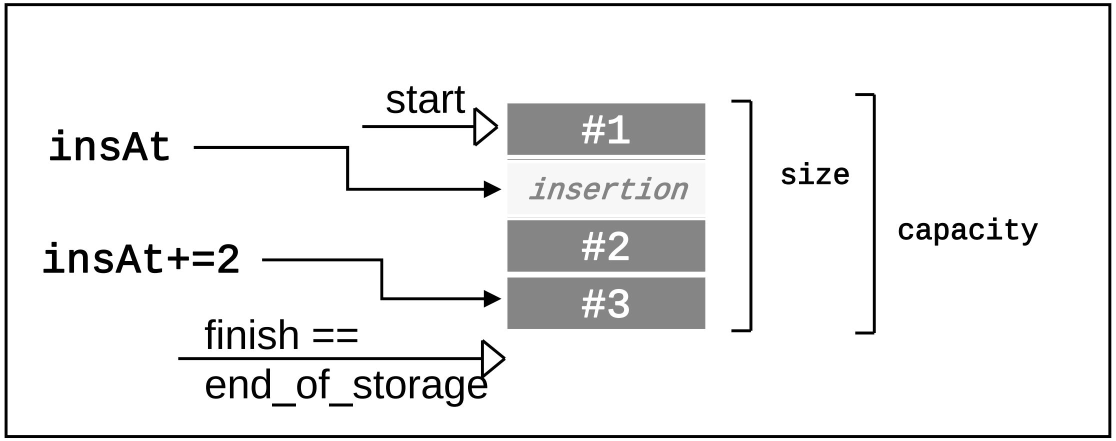
Consider a program that
inserts a line at a specific position, say, before any line that starts with a capital letter.

# Inserting elements to a vector

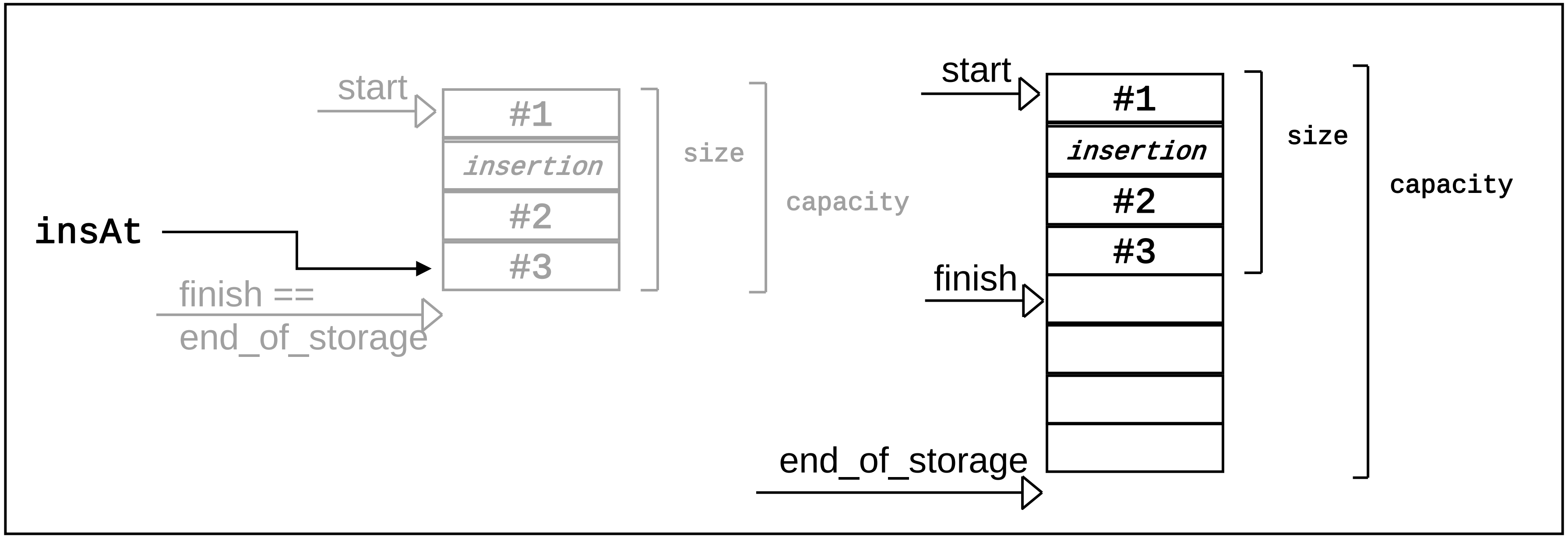
```
void doIt(const char* in,const char* out)
{  vector<string> buf;
   // ... populate vector ...

   vector<string>::iterator insAt = buf.begin();
   while ((insAt = find_if(insAt,buf.end(),isUpper()))
          != buf.end())                 <<<<<< crash !!!

   {   buf.insert(insAt,"text to be inserted");
       insAt+=2;
   }
}
```

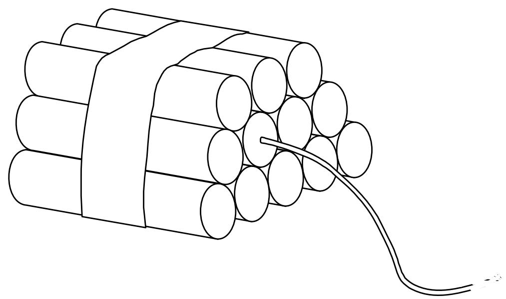
# Inserting elements to a vector



# Inserting elements to a vector



# Back to the initial problem ...

```
void doIt(const char* in,const char* out)
{  vector<string> buf;
   vector<string>::iterator insAt = buf.end();
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.insert(insAt,linBuf);

   sort(buf.begin(),buf.end());

   ofstream outFile(out);
   copy(buf.begin(),buf.end(),
        ostream_iterator<string>(outFile,"\n"));
}
```

**<<<<<< crash !!!**

# A suggested solution

`insert()` returns an iterator to the newly inserted element.

Use this new, valid position as the point of insertion for subsequent insertions.

```
void d(Itconst char* in,const char* out)
{  ...

   while(getline(inFile,linBuf))
         insAt = buf.insert(insAt,linBuf);

   ...
}
```

# Another Solution

More elegant and easier to comprehend is the use of the `push_back()` function instead of the `insert()` function.

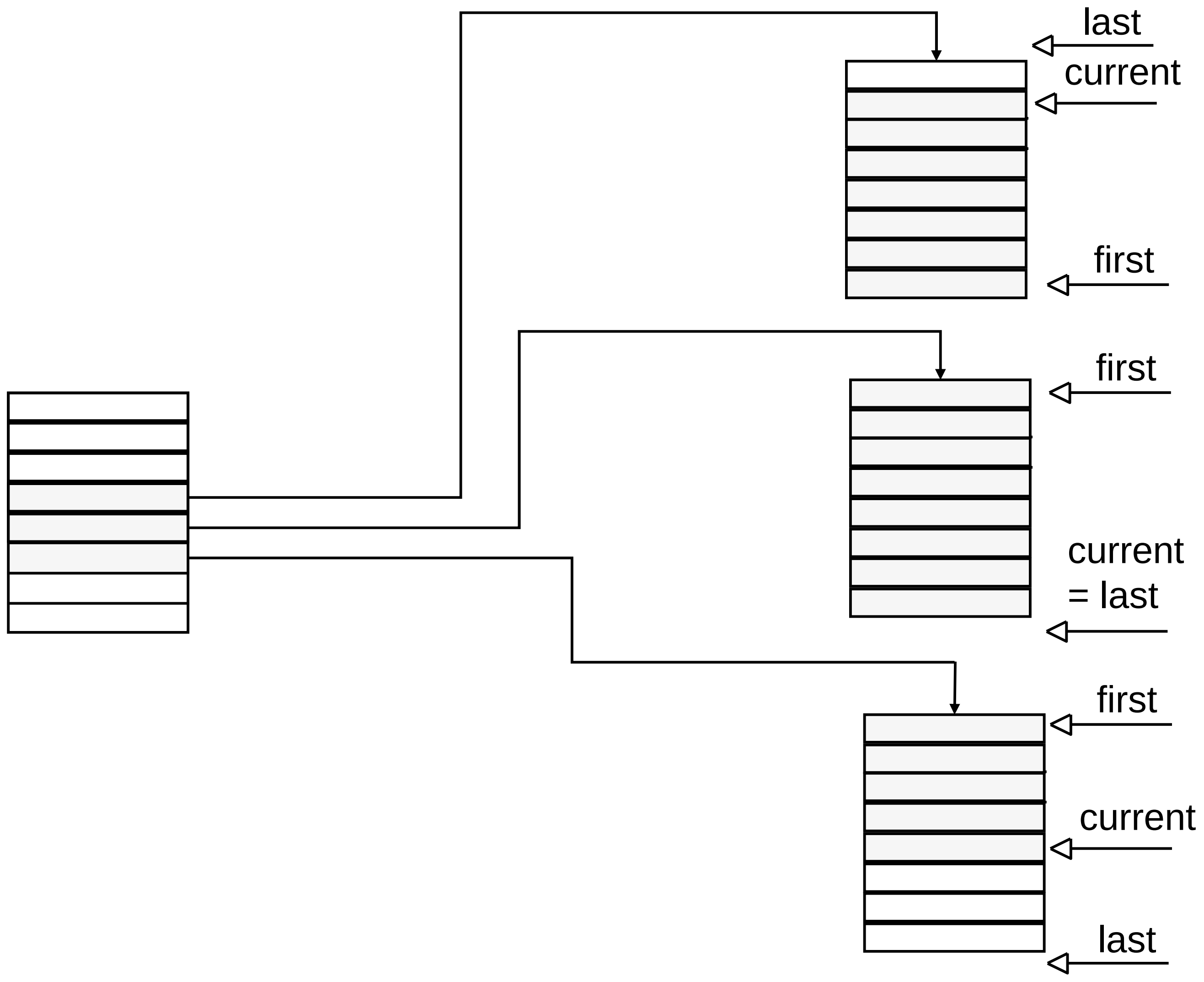
```
void doIt(const char* in,const char* out)
{
   deque<string> buf;
   string linBuf;
   ifstream inFile(in);

   while(getline(inFile,linBuf))
         buf.push_back(linBuf);


   ...
}
```

# Not yet considered ...

Validity guarantee for insertion into a deque:

- An insert in the middle invalidates all the iterators and references.
- An insert at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references.

# A typical deque implementation



# Validity guarantee for erase()

For associative containers and list:

- Invalidates only iterators and references to the erased elements.

For vector:

- Invalidates all the iterators and references after the point of the erase.

For deque:

- An erase in the middle invalidates all the iterators and references to elements of the deque.
- An erase at either end invalidates only the iterators and the references to the erased elements.

# STL Pitfall #2

## Mutable or Immutable Set Iterators ?

# A set container in the STL ...

- is implemented as a binary tree
- needs a strict weak ordering for the elements
- allows insertion and removal of elements via `insert()` and `erase()`, which use the ordering for maintaining the tree structure
- provides iterators that give access to the contained elements

Note:

- Elements must not be modified through an iterator because direct manipulation of the elements would corrupt the tree structure.
- A set implementation need not provide a mutable iterator.

# Conceivable implementations

Details are still an open issue (#103 on the library issue list of August 1999). Two implementations for set iterators are conceivable:

constant iterator:

- fool-proof: no chance to modify the elements in-place
- restrictive: cannot change parts of the element that do not contribute to the ordering

mutable iterator:

- security hole: can inadvertently corrupt the tree structure

Consider a program that
implements a bank account class,
creates a set of bank accounts, and
tries to assign to an element through an iterator.

# Modification through a set iterator

```
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; }
```

```
set<account> s;
…
set<account>::iterator iter;
…
*iter = *new account;  // direct modification of element
```

# Obvious mistake

```
set<account>::iterator iter;
…
*iter = *new account;  // direct modification of element
```

Overwriting an element in the tree structure is likely to destroy the structure.

Result:

constant iterator:

– error message; will not compile

mutable iterator:

– will corrupt the tree structure; subsequent behavior is unpredictable

# Suggested solution

```
set<account>::iterator iter;
…
*iter = *new account;  // direct modification of element
```

Never "replace" an element in a set;
insert the new one and erase the old one.

```
set<account>::iterator iter;
…
s.insert(iter, *new account);
s.erase(iter);
```

# Modification through a set iterator

```
class account {
 …
 size_t _number;  // determines ordering
 double _balance; // irrelevant for ordering
};

bool broke(const account& acc)
{ return acc.balance() <= 0; }
```

```
set<account> s;
…
// remove element if balance is 0 or less
set<account>::iterator garbage;
garbage = remove_if(s.begin(),s.end(),broke),s.end();
s.erase(garbage,s.end());
```

# A less obvious mistake

```
set<account>::iterator iter;
…
garbage = remove_if(s.begin(),s.end(),broke),s.end();
```

`remove_if()` is a mutating algorithm, that is, it performs in-place modifications on the container elements via the iterator.

Result:

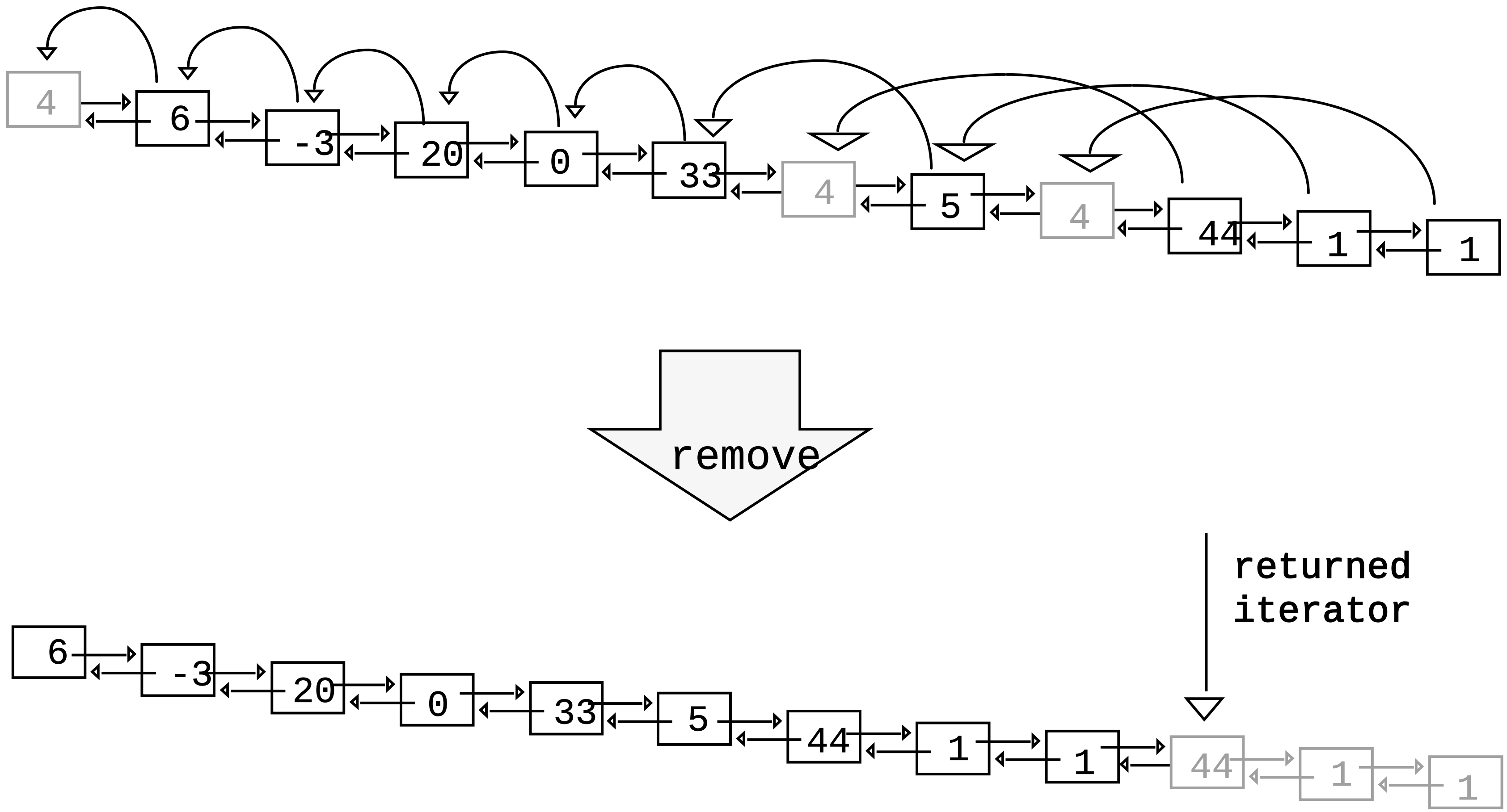
constant iterator:

- error message; will not compile
- warning: "discards const"; will corrupt the tree structure

mutable iterator:

- will corrupt the tree structure; subsequent behavior is unpredictable

# The remove() algorithm



# Suggested solution

```
// remove element if balance is 0 or less
set<account>::iterator garbage;
garbage = remove_if(s.begin(),s.end(),broke),s.end();
s.erase(garbage,s.end());
```

Never apply mutating algorithms to a set;
instead of (mutating) `remove_if()`
use inspecting) `find_if()`.

```
set<account>::iterator fnd;
for (fnd=find_if(s.begin(),s.end(),broke)
    ;fnd!=s.end()
    ;fnd=find_if(fnd,s.end(),broke))
{   s.erase(fnd++);    }
```

# Modifiying part of the element

```
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; }
```

```
set<account> s;
…
set<account>::iterator iter;
…
// direct modification of part of the element
iter->balance = 1000000;
```

# Not at all a mistake

```
set<account>::iterator iter;
…
// direct modification of part of the element
iter->balance = 1000000;
```

The balance does not contribute to the ordering relationship.
Modifications of the balance would not affect the tree structure.

Result:

constant iterator:

– error message; will not compile

mutable iterator:

– works nicely

# Conceivable solutions

If the set iterator does not allow modification of the insignificant part of the element:

- Cast away constness.
- Provide a const member function in class `account` that performs the desired modification.
- Implement an iterator adapter that allows the desired modification.

# The Brute Force Approach

```
set<account>::iterator iter;
…
// direct modification of part of the element
iter->balance = 1000000;
```

Cast away constness:

```
set<account>::iterator iter;
…
// direct modification of part of the element
*(const_cast<double*>(&(iter->_balance))) = 1000000;
```

# A little more sophisticated

Encapsulate the cast into a const member function of the `account` class:

```
class account {
public:
 void setBalance(double b) const
 { *const_cast<double*>(&_balance) = b; }
 …
private:
 size_t _number;  // determines ordering
 double _balance; // irrelevant for ordering
};
…
iter->setBalance(1000000);
```

Safety hole: can also modify constant objects of type `account`

# An iterator adapter

```
set<account>::iterator iter;
…
// direct modification of part of the element
iter->balance = 1000000;
```

Define an iterator adapter `balanceIter` that adapts the set iterator.

Its dereference operator returns a non-const reference to the balance of the element pointed to.

```
set<account>::iterator iter;
…
// direct modification of part of the element
*balanceIter(iter) = 1000000;
```

# A simple iterator adapter

```
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;
};
```

# The 3 suggested solutions

- Casting away constness is the brute force approach; it can and should be avoided.
- Providing a const member function that performs the modification is error-prone; allows modification of const objects.
  - Not a viable solution if the implementation of the account class must not be changed.
- The iterator adapter is the most flexible solution:
  - cast is safely encapsulated;
  - no change to the account class necessary;
  - no security hole; cannot change elements through constant iterators
  - can apply algorithms to the adapted iterator

# Conclusions

- It's unfortunate that details of the set iterator are still an open issue.
  - impairs portability efforts
- Avoid direct access through a set iterator to any part that is significant to the ordering of elements.
  - Never allow lvalue use of a dereferenced set iterator.
- Avoid applying mutating algorithms to sets.
  - Read the damned manual.
- If you want to modify a non-significant part and need to get around the const-restriction, build proper abstractions.
  - see for instance the iterator adapter

# STL Pitfall #3

Function objects
must not have
side effects

# Function objects in the STL ...

- are accepted as arguments to numerous algorithms
- can be function pointers of functors
- must not have side effects
- must not modify container elements through an iterator

Consider a program that

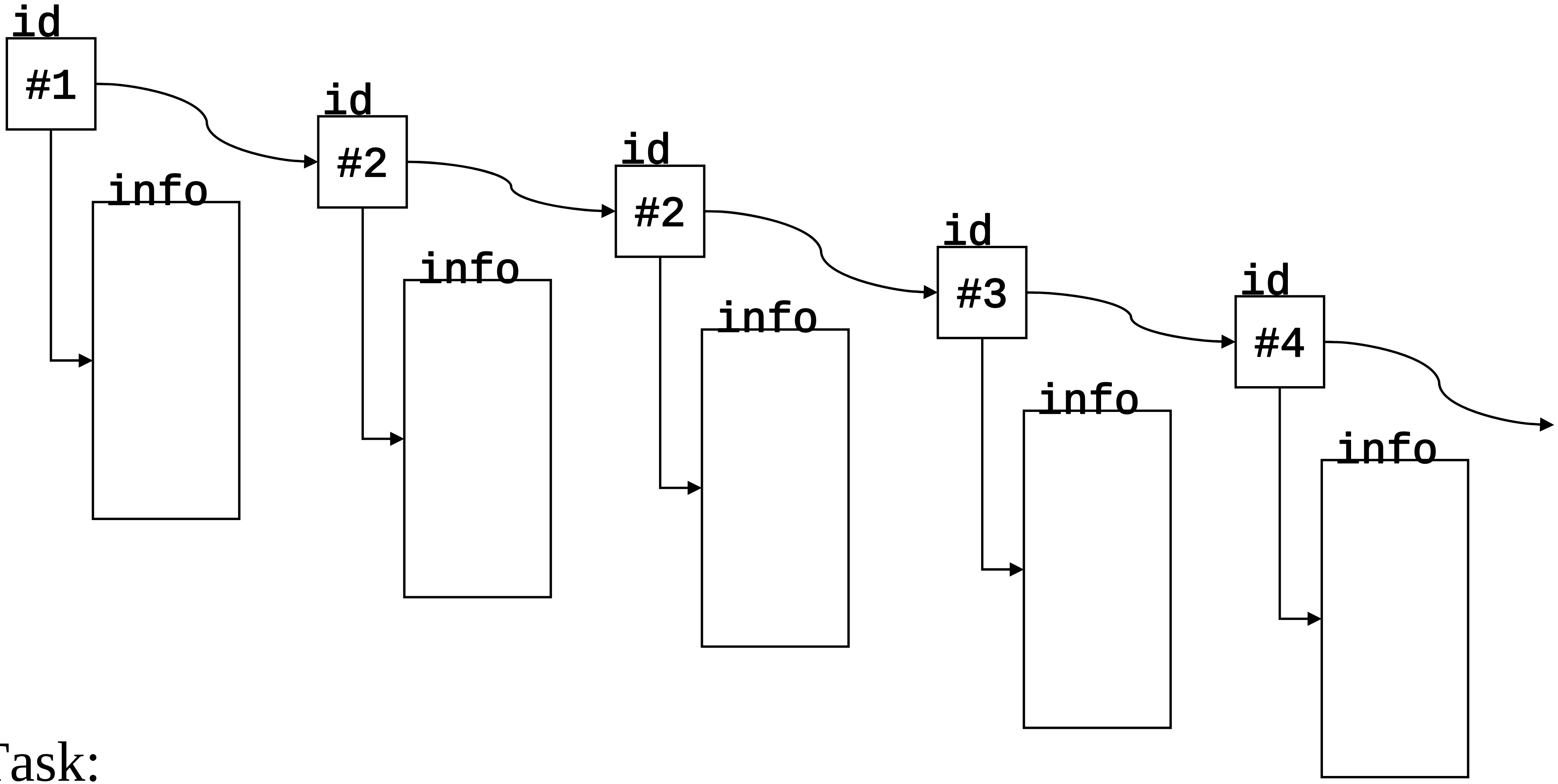
removes duplicates (with the same id) from a container,
accumulates any information associated with the id,
defines a functor to perform the compression, and
calls `unique()` passing the functor to get the job done.

# An insurance application

```
struct accident {
   string owner;   string insurance;
   string date;    bool   dumped;
};
struct insuranceRec {
   insuranceRec(long id, const list<accident>& c);
   long vehicleId;
   list<accident> crashes;
};
```

```
multiset<insuranceRec> clients;
// ... populate container ...
clients.erase(  // clean up: remove duplicates
   unique(clients.begin(), clients.end(), mergeRec()),
   clients.end());
```

# Clean up

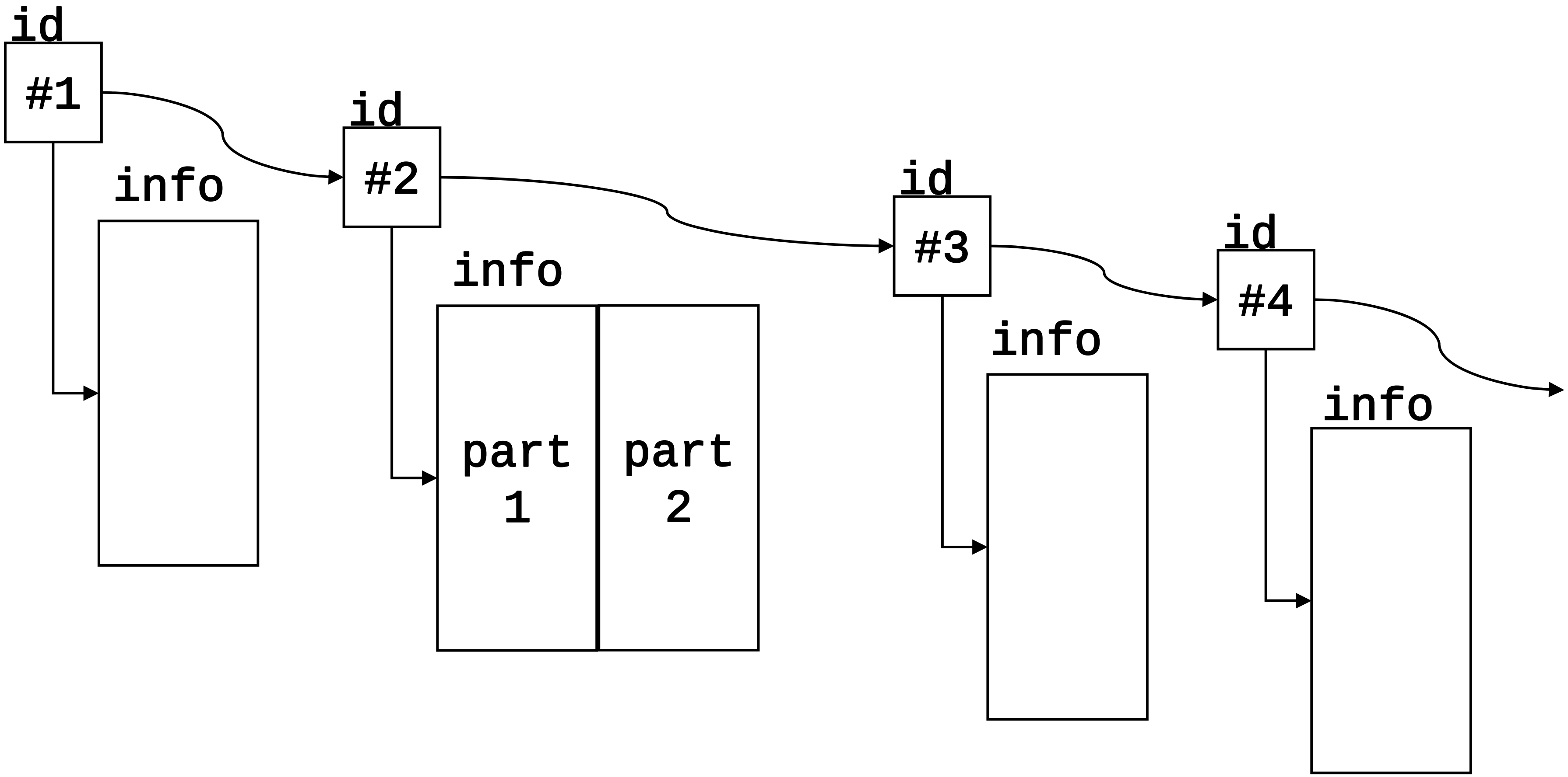



Task:

•remove duplicates (with the same id)

•merge the associated information into the remaining entry

# Clean up



# Implementing the clean up

We want to use the `unique()` algorithm for elimination of consecutive duplicates:

```
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last,
          BinaryPredicate pred);
```

We need to define a predicate that

- determines the duplicates (i.e. it must check for identical ids) and
- produces a side effect (i.e. merging the associated info)

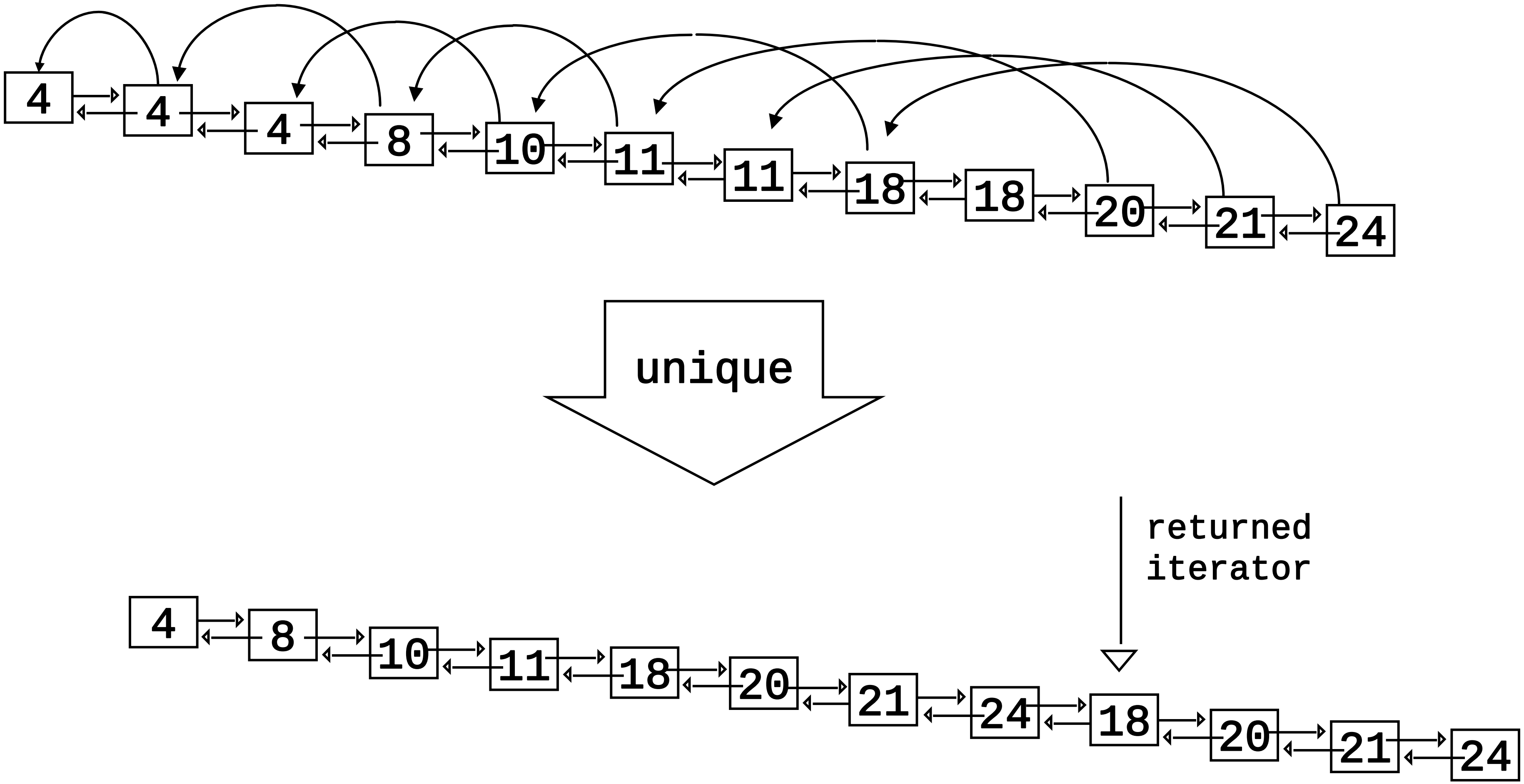
# The function object type

```
class mergeRec {
public:
   bool operator()(      insuranceRec& lhs,
                   const insuranceRec& rhs)
   { // predicate: check for same id
     bool sameId = (lhs == rhs);

     if (sameId)
        // produce side effect: append rhs-info to lhs-info
        copy(rhs.crashes.begin(), rhs.crashes.end(),
             inserter(lhs.crashes,lhs.crashes.end()));

     return sameId;
   }
};
```

# The element type

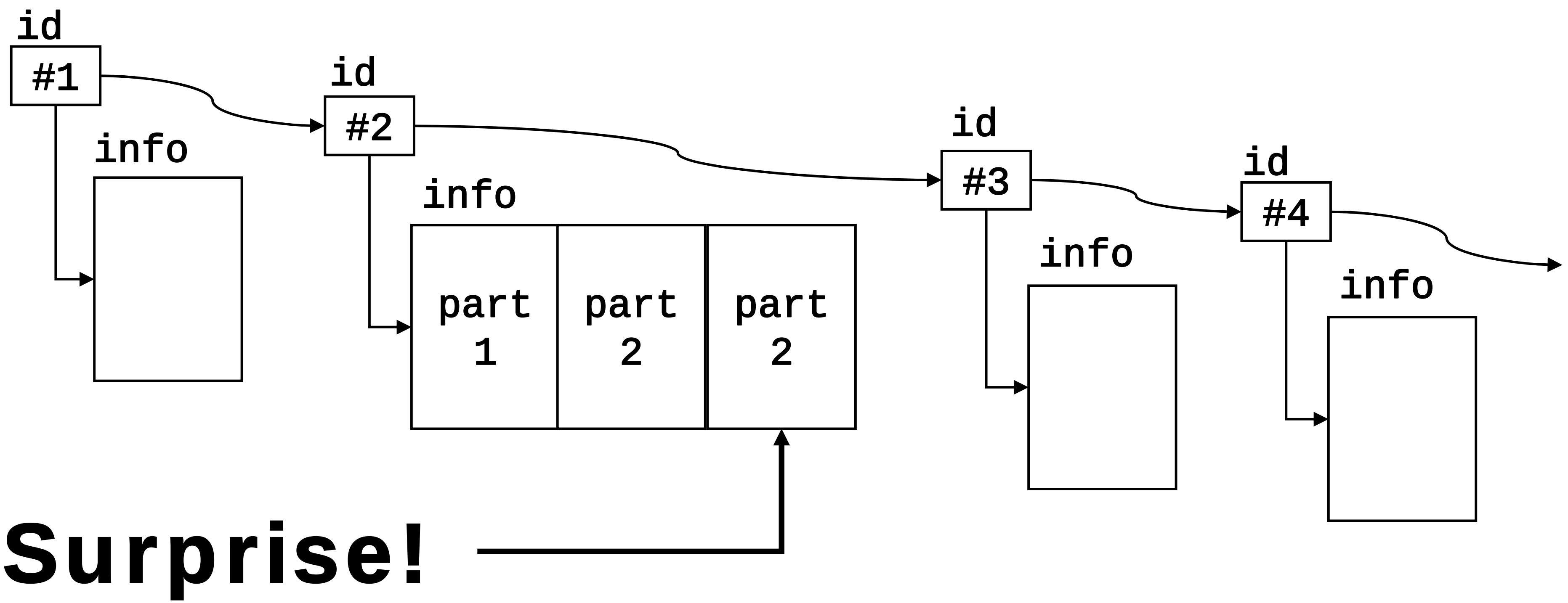
```
struct accident {
   string owner;   string insurance;
   string date;    bool   dumped;
};

struct insuranceRec {
   insuranceRec(long id, const list<accident>& c);
   long vehicleId;
   list<accident> crashes;
};

bool operator==(const insuranceRec& lhs,
                const insuranceRec& rhs)
{ return lhs.vehicleId == rhs.vehicleId; }
```

# The unique() algorithm



# Doing the clean up

```
multiset<insuranceRec> clients;
// ... populate container ...
clients.erase(  // clean up: remove duplicates
   unique(clients.begin(), clients.end(), mergeRec()),
   clients.end());
```



# What happened?

- Our predicate does not only check for the duplicates but in addition produces a side effect (i.e. merging the associated info).

  Function objects must not have side effects and must not modify any element through an iterator.

- Both is violates, which does not always create problems, but in our case we do not know how often the `unique()` algorithm produces the predicate's side effect.

  It is unspecified how often an algorithm invokes a function object on the same (pair of) element(s).

# Implementation of unique()

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last,
       BinaryPredicate binary_pred)
{
   first = adjacent_find(first, last, binary_pred);
   return unique_copy(first, last, first, binary_pred);
}
```

`adjacent_find()` return the first element of a series of duplicates.

`unique_copy()` copies all elements except for consecutive duplicates.

# Conceivable solutions

- Do not provide functions (or function objects) that produce side effects to any STL algorithm.
  - implement side effects separately
  - use STL algorithms only for side-effect-free operations

- Implement your own version of unique() so that you have control over the number of side effects produced.

# Side-effect-free predicate

Instead of calling `unique()` with our `mergeRec` predicate

- call `adjacent_find()` with a side-effect-free predicate `eqRec` and `equal_range()` to identify any duplicates and
- produce the side effect independently of the STL.

```
class eqRec {
public:
   bool operator()(const insuranceRec& lhs,
                   const insuranceRec& rhs)
   { return (lhs == rhs); }
};
```

# The actual logic

```
multiset<insuranceRec> clients;
// ... populate container ...
typedef multiset<insuranceRec>::iterator iterType;
iterType duplicate = clients.begin();
while (duplicate!=clients.end())
{   // identify duplicates
    duplicate =
       adjacent_find(duplicate,clients.end(),eqRec());

    // merge info and erase duplicates
    if (duplicate!=clients.end())
    { pair<iterType,iterType> range;
      range = clients.equal_range(*duplicate);
      compress(range);
      clients.erase(++(range.first),range.second);
    }
}
```

# Producing the side-effect

```
template <class Iterator>
void compress(pair<Iterator,Iterator> range)
{
   range.second--;
   copy(range.second->crashes.begin(),
        range.second->crashes.end(),
        inserter(range.first->crashes,
                 range.first->crashes.end())
       );
}
```

# Customizing unique()

Instead of calling the STL `unique()` implement your own version in order to gain control of the side effects produced.

- pro:
  - portable, reusable
  - potentially more efficient
- con:
  - extra effort required

# User-defined version of unique()

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator
my_unique(ForwardIterator first, ForwardIterator last,
          BinaryPredicate binary_pred)
{ if (first == last)  return last;
  else { ForwardIterator next = first;
         while(++next != last)
         { if (!binary_pred(*first, *next)) first = next;
           else // duplicate found
           {  while (++next != last)
              if (!binary_pred(*first, *next))
                 *++first = *next;
              return ++first;
           }
         }
         return last;
       }
}
```

# Evaluation

- using STL version of `unique()` with a side-effect producing predicate
  - elegant and readable, but non-portable
- using customized version of `unique()` with a side-effect producing predicate
  - still elegant and readable, but also portable
  - extra effort required
- strict separation between side-effect-free predicate and a side-effect producing function
  - comparable to customized unique() regarding effort and complexity
  - portable, but probably not reusable

# Further problematic cases

- function objects that modify any part of an element that is relevant to the ordering
  – There is no mechanism to make sure that only immutable iterators are provided to a function object.
  – Hence you can change whatever you like - even corrupt the container.
- function objects that depend on how often they are invoked
  – The number of produced side effects is not specified.
  – Any side-effect-producing function object falls into this category.
- function objects that depend on how often they are copied, assigned, or destroyed
  – It is unspecified how many temporary copies of a function object ara algorithms creates.
  – All function objects that have non-constant state and accumulate data between subsequent invocations fall into this category.

# Function objects with state

Example:

- Our side-effect producing predicate might make a note (in an internal list) of all duplicates that it finds.

Problem:

- Hardly any algorithm returns the function objects that it received.
  - How do I get hold of the accumulated data?
  - The predicate writes the entire information to a global or static location when it is destroyed.

# References to function objects

`for_each()` is the only algorithm that returns the function object.

An alternative approach for getting access to the object state:

- Pass the function objects by reference rather than by value.
  - Requires explicit function arguments specification syntax and creates lifetime dependencies.

```
unique(clients.begin(), clients.end(), mergeRec());
```

would become

```
mergeRec predicate;
unique<mergeRec&>(clients.begin(), clients.end(),
                  predicate);
```

# Destructor with side effects

The predicate writes the entire information to a global or static location when it is destroyed.

- The destructor has a side effect.
- It is unspecified
  - how many temporary copies of the function object are created inside `unique()` and
  - how often the destructor is invoked.

Rule:

- Never create functions objects that produce side effects when they are created, copied, or destroyed.
- This is common sense for any class, but even more important for types that are provided to the STL.

# Implementation of unique()

```
template <class ForwardIterator, class BinaryPredicate>
ForwardIterator
unique(ForwardIterator first, ForwardIterator last,
       BinaryPredicate binary_pred)
{
   first = adjacent_find(first, last, binary_pred);
   return unique_copy(first, last, first, binary_pred);
}
```

We cannot tell how often the predicate is copied unless we also study the implementations of `adjacent_find()` and `unique_copy()`.

# Restrictions to function objects

- Never create functions objects that produce side effects when they are created, copied, or destroyed.
  - This is common sense for any  class, but even more important for objects that are provided to the STL.
- Be careful with function objects that create side effect when they are invoked.
  - It is not at all uncommon that functions have side effects.
- Be careful with function objects that modify the elements through the iterator.
  - It is not at all uncommon that functions which take pointers/references modify the pointed to objects, and iterators have pointer-like semantics.

# STL Pitfall #4

## Comparators must not be polymorphic

# Polymorphic comparators

- It's a common technique to implement a family of compare policies as a class hierarchy with a common base class and to invoke the policies through base class reference for polymorphic behavior. (See the Strategy pattern à la GOF.)

- The associative containers accept a reference to a comparator object; hence one could pass a base class reference.

- However, they store a copy of the comparator object internally, the result of which is object slicing.

# STL Pitfall #5

## Equality vs. induced equivalence

# Equality vs. equivalence

- The associative containers use an induced equivalence relation for maintaining the underlying tree structure.
- The equivalence is educed from the ordering, i.e., the comparator that is provided to the container.
- Container member functions use the equivalence relations for finding equal elements in the container.

- STL algorithms use an equality relation on the iterator's value type for finding equal elements.

# Equality vs. equivalence

- Can lead to surprising results when equality and equivalence are different.

Example: case-insensitive string compare

- Strings that are equivalent (regarding case) are not necessary equal.
- Consider a multiset of strings with case-insensitive order in conjunction with “set” algorithms such as `union()` or `intersection(` .

# STL Pitfall #6

Type incompatibility of adapted iterators

# Adapted iterators

- STL algorithms accept any kind of iterator type, because they are function templates.
- Container member functions only accept their own iterator type.

Example:

- A container element is searched for by passing reverse iterators to `find()`.
- The resulting iterator cannot be passed to the container's `erase()` function.
- When implementing an iterator adapter, never forget ot implement the `base()` member function.

# Several stream iterators on the same stream

# Interdependent stream iterators

- Stream iterators on the same stream are not independent of each other.
- Advancing one iterator affects all other iterators on the same stream, because it changes the underlying stream position.

- For input stream iterators:
  - Increment means reading from the stream.
  - Dereferencing means providing the stored, previously read value.
- For output stream iterators:
  - Increment and dereferencing are NOPs.
  - Assignment to the iterator means writing to the stream.

# Interdependent stream iterators

- Reaching the end iterator means failure of the read/write operation, i.e. reaching end of input or an error situation.

Example: 2 input stream iterators for reading a string and a float

- Reading a string also moves the float iterator.
- Reading a float when there is a string on the file turns the float iterators into an end iterator; the string iterator can still read.
  - The frozen float iterator cannot be reset.
  - The stream state must be cleared and a new float iterator must be created.

# STL Pitfall #8

Allocators must exhibit static behavior

# Static allocators

Allocators of the same type must be interchangeable,
i.e. must not have state.

Reason:

- Two containers of the same type, i.e. using the same type of allocator, can have different allocator objects.
  - Example: database allocators to different databases
- If the two containers are assigned to each other, all elements must be copied. Which allocator must be used?
- There’s no universal answer.
- The problem evaporates when all allocators of the same exhibit the same behavior.

# References

**Generic Programming and the Stl**

Matthew H. Austern

Addison Wesley Longman, 1998

**The C++ Standard Library**

Nicolai M. Josuttis

Addison Wesley Longman, 1999

**C++ Report** (SIGS Publications) Columns

- Effective Standard Library - Klaus Kreft & Angelika Langer
- Sutter's Mill- Herb Sutter
- The (B)leading Edge - Jack Reeves

# Contact Info

**Angelika Langer**

Training & Mentoring
Object-Oriented Software Development in C++ & Java

email: **langer@camelot.de**
http: **//www.camelot.de/~langer/**