why talk about invalid iterator?

- iterators are a fundamental concept in the STL
  - play an important role as glue between containers and algorithms
- only valid iterators yield predictable results
  - invalid iterators should never be used
- in practice we make mistakes
  - invalid iterators are used inadvertently

- knowledge about invalid iterators aids:
  - identifying and avoiding invalid iterators
  - tracking down bugs caused by invalid iterators
agenda

• valid iterators and iterator ranges

• invalid iterators
  – singular iterators
  – past-the-end iterators
  – out-of-range iterators
  – dangling iterators
  – inconsistent iterators

what is an iterator?

• generalized pointer:
  – gives access to all elements in a sequence

  – required operations:
    • dereferencing operator ( * p )
    • incrementing operator ( p ++ )
    • comparison operator ( p == q )
**iterators = generalized pointers**

```cpp
template <class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value) {
    while (begin != end && *begin != value) {
        begin++;
    }
    return begin;
}
```

**combining containers and algorithms**

Compare “iterators provided” to “iterators required”:

- A container description includes the strongest iterator categories it provides.
- An algorithm description includes the weakest iterator categories it requires.
### Iterators in the STL

<table>
<thead>
<tr>
<th><em>iterator</em></th>
<th><em>iterator concept</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer to array</td>
<td>random access</td>
</tr>
<tr>
<td>iterator to vector / deque</td>
<td>random access</td>
</tr>
<tr>
<td>iterator to list</td>
<td>bidirectional</td>
</tr>
<tr>
<td>iterator to (multi)set / (multi)map</td>
<td>bidirectional</td>
</tr>
<tr>
<td>iterator to input stream</td>
<td>input</td>
</tr>
<tr>
<td>iterator to output stream</td>
<td>output</td>
</tr>
<tr>
<td>insert iterator</td>
<td>output</td>
</tr>
</tbody>
</table>

### Validity

- **Valid iterators**
  - can be **advanced**, **dereferenced** and **compared**
  - more precisely:
    - support all operations of their iterator category

- **Valid iterator range**
  - consists of valid iterators (**beginning** and **past-the-end**)
  - end iterator must be **reachable**
valid iterators - examples

```cpp
istream_iterator<string> beg(cin), end;
vector<string> vec(beg, end);
list<string> lst;
copy(vec.begin(), vec.end(),
    front_inserter(lst));
copy(lst.begin(), lst.end(),
    ostream_iterator<int>(cout, "\n");
```

subtle bugs - invalid iterators

```cpp
istream_iterator<string> beg(ifstream("in.txt")), end;
copy(beg, end, ostream_iterator<int>(ofstream("out.txt")));
vector<string> vec(beg, end);
list<string> lst;
list<string>::iterator outlier;
copy(vec.begin(), vec.end(), outlier);
copy(vec.begin(), vec.end(), lst.begin());
```
subtle bugs - invalid iterators

```cpp
ifstream inFile("in.txt");
istream_iterator<string> beg(inFile), end;
copy(beg, end, ostream_iterator<int>(cout));
vector<string> vec(beg, end);
copy(vec.begin(), vec.end(),
     ostream_iterator<int>(cout));
```

subtle bugs - invalid iterators

```cpp
istream_iterator<int> beg(cin), end;
vector<int> vec(beg, end);
vector<int>::iterator iter = ... some interesting position ... ;
for (int n=1; n<=100; ++n)
    vec.insert(iter, n);
vec.erase(remove(vec.begin(), vec.end(), 0),
         vec.end());
cout << *iter << endl;
```

might be inconsistent (or dangling) iterator

might turn into dangling iterator
invalid iterators

golden rule #1: Never use invalid iterators.

- result of using invalid iterators is undefined
- expressions such as \*\texttt{iter}, ++\texttt{iter}, etc.
  - exhibit “undefined behavior”
    - which can be anything
    - from returning a valid and useful result
    - to a program crash or reformattting of your hard-disk

agenda

- valid iterators and iterator ranges
- “invalid” iterators
  - singular iterators
  - past-the-end iterators
  - out-of-range iterators
  - dangling iterators
  - inconsistent iterators
**singular iterators - definition**

quote from the standard:

*Iterators can have singular values that are not associated with any container.*

*Results of most expressions are undefined for singular values; the only exception is an assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value.*

**singular iterators - examples**

- uninitialized pointers
  ```cpp
  int* ptr;
  ```

- default-constructed container iterators
  ```cpp
  list<int>::iterator iter;
  ```

- default-constructed iterator adapters
  ```cpp
  reverse_iterator<int*> rIter;
  ```

- dereferenceable and past-the-end values are non-singular
  - example: default-constructed input stream iterators
    ```cpp
    istream_iterator<int> eof;
    ```
why do we care?

- singular iterators can be created
- can be used inadvertently as input or output iterators

example:

```cpp
int array[100];
int* begin, end;
list<int> lst;
list<int>::iterator out;

copy(begin, end, out);
```

singular iterators

- are not associated with any container
- only assignment is defined
  - results of most expressions are undefined for singular iterators
  - only assignment of a non-singular iterator to a singular iterator is valid

Never perform any operation on a singular iterator except assignment of a non-singular iterator.

golden rule #2:
agenda

- valid iterators and iterator ranges
- "invalid" iterators
  - singular iterators
  - past-the-end iterators
  - out-of-range iterators
  - dangling iterators
  - inconsistent iterators

past-the-end iterators - definition

quote from the standard:

Just as a regular pointer to an array guarantees that there
is a pointer value pointing past the last element of the
array, so for any iterator type there is an iterator value
that points past the last element of a corresponding
container. These values are called past-the-end values.

Values of an iterator \( i \) for which the expression \( *i \)
is defined are called dereferenceable. The library never
assumes that past-the-end values are dereferenceable.

additional requirement in the standard:

Iterators that can be incremented must be dereferenceable.
past-the-end iterators - examples

- non-dereferenceable past-the-end iterators
  - end-of-container iterator `container.end()`
  - end-of-array iterator `array+size`
  - end-of-input-stream iterator `istream_iterator<T>()`
  - reverse past-the-end iterator `container.rbegin()`
  - reverse end-of-array iterator `reverse_iterator<elemT*>(array)`

- dereferenceable past-the-end iterator:

```cpp
int arr[500];
...
int* where = find(arr, arr+100, 5);
```

why do we care?

- past-the-end iterators can be created
- can be used inadvertently as input or output iterators

example:

```cpp
int array[100];
list<int> lst;
copy(array, array+100, lst.begin());
```

- list is empty
  ⇒ begin iterator equals end iterator
invalid operations inside algorithm

```cpp
template <class In, class Out>
Out copy ( In first, In last,
Out result)
{
    while (first != last)
        *result++ = *first++;
    return result;
}
```

valid operations

- past-the-end iterators support all operations of their respective iterator category
  - except dereferencing and increment

```cpp
pastTheEnd-- or pastTheEnd_N
```

- valid for a bidirectional or random-access iterator
  - example: `list.end()--` or `vector.end() - 1`

```cpp
pastTheEnd_begin
```

- distance can be calculated for a random-access iterators
  - example: `vector.end() - vector.begin()`

```cpp
insert(pastTheEnd, value)
```

- insertion before past-the-end iterator is allowed
  - example: `container.insert(container.end(), value)`
valid operations - example

```cpp
iostream_iterator<int> in(cin), eof;
vector<int> vec(in, eof);
sort(vec.begin(), vec.end());
cout << *(vec.begin()) << " \t " << *(vec.end()-1);

vector<int>::iterator pos;
pos = lower_bound(vec.begin(), vec.end(), VALUE);
vec.insert(pos, VALUE);
```

past-the-end iterators

- point past the last sequence element
  - used as end of an iterator range

- might be non-dereferenceable and non-incrementable
  - expressions \*iter and ++\*iter might be invalid
  - no algorithm dereferences or advances a past-the-end iterator

golden rule #3: Never dereference or increment the past-the-end iterator of an iterator range.
null
**why do we care?**

- out-of-range iterators can inadvertently be created
  - often implicitly inside an algorithm
- all operations are invalid, yet they might work somehow
  - knowledge of their behavior aids bugs tracking

**example:**

```c++
istream_iterator<string> in(cin), eof;
vector<string> vec; vec.reserve(100);
copy(in, eof, vec.begin());
```

- algorithm might advance iterator beyond capacity
- unpredictable result
  - memory corruption w/o program crash

---

**common situation in the STL**

- out-of-range iterators can be created inadvertently
  - whenever size of sequence is determined by information other than the sequence itself

- examples:
  - all algorithms that take output iterator
    - size of output sequence determined by size of input sequence
    - `copy()`, `remove_copy_if()`, `transform()`, `merge()`, ...
  - algorithms with more than one input sequence
    - size of 2nd input sequence determined by size of 1st input sequence
    - `binary_transform()`
**typical implementation of vector**

non-empty vector

![Diagram of vector implementation]

- **start**
  - Al
  - Alberto
  - Alesa
  - Amr
  - Amy
  - Andy
- **end_of_storage**
- **size**
- **capacity**

**example using vector**

```cpp
istream_iterator<string> in(cin), eof;
vector<string> vec; vec.reserve(100);
copy(in, eof, vec.begin());
```

- in our example the vector is empty, but has memory reserved (size: 0, capacity: 100, begin == end)
- `copy()` overwrites reserved positions until capacity is exhausted and crashes then
- vector remains empty, although elements have been overwritten
  - internals such as size, capacity, begin, end are only modified via container operations, never through iterators
**vector before/after copy()**

- `start == finish
- Al
- Alberto
- Alesa
- Amr
- Amy
- Andy
- size: 0
- capacity: 100
- end_of_storage

**example using empty vector**

```cpp
istream_iterator<string> in(cin), eof;
vector<string> vec; // empty vector
copy(in, eof, vec.begin());
```

- if vector is empty and has no memory reserved
  
  (size: 0, capacity: 0, begin == end == 0)

  `⇒` immediate crash

empty vector

- nothing allocated
- all pointers are null pointers
- size and capacity are zero

```
start
== finish
== end_of_storage
== 0
```

```
size
== capacity
== 0
```

VS .NET Connections
**recommendation**

- avoid problem: use inserters as output destination
  - insert iterators have no valid range
  - can be incremented infinitely often

```
istream_iterator<string> in(cin), eof;
vector<string> vec;
copy(in, eof, back_inserter(vec));
```

**golden rule #4:**

Prefer inserters as output destinations over “regular” iterators.

**example using non-empty list**

```
istream_iterator<string> in(cin), eof;
list<string> lst;
// fill and use list
// re-fill by overwriting
copy(in, eof, lst.begin());
```

- assume more input than `lst.size()`, i.e. list iterator advanced beyond end
- possible result: 
  - [GNU] / [CW] cyclic overwriting of list elements
  - no immediate crash, list corrupted unexpected content, crashes later
- even more confusing with read-access to out-of-range positions
  - no crash; infinite cycle over list elements
**typical implementation of list**

- non-empty list

![Diagram of list implementation](image)

**example using set (after end)**

```cpp
istream_iterator<string> in(cin), eof;
ostream_iterator<string> out(cout, "\n");
multiset<string> mset(...);  // non-empty set
transform(in, eof, mset.begin(), out, plus<string>());
```

- assume, algorithm advances set iterator beyond end
- possible result:
  - [GNU] oscillates (end ↔ end-1) ⇒ no crash
  - [CW] immediate crash ⇒ crash

- crashes if out-of-range positions are overwritten
  - modification destroys sorting order and corrupts tree structure
  - some implementations do not provide write iterators for (multi)set
**typical implementation of set**

- non-empty set

---

**example using** `istream_iterator`  

```cpp
istream_iterator<string> in(cin), eof;  
ifstream fil("in.txt");
copy(in, eof, istream_iterator<int>(fil));
```

- assume, algorithm advances stream iterator beyond the end
- result depends on implementation of stream iterator
- possible result:  
  - [GNU] freezes at end ⇔ no crash
  - [CW] crashes at end  ⇔ crash
GNU implementation of `istream_iterator`

```cpp
template <class elemT> class istream_iterator {
protected:
    istream* stream; bool end_marker; elemT value;
    void read() { 
        end_marker = (*stream) ? true : false; 
        if (end_marker) *stream >> value;
    } end_marker = (*stream) ? true : false;
public:
    istream_iterator() : end_marker(false) {} 
    istream_iterator (istream& s) : stream(&s) { read(); } 
    const elemT& operator*() const {return value;} 
    istream_iterator<elemT>& operator++() 
    { read(); return *this; }
};
```

will freeze
if out of range

Metrowerks implementation

```cpp
template <class elemT> class istream_iterator {
private:
    istream* stream; elemT value;
public:
    istream_iterator() : stream(0) {} 
    istream_iterator (istream& s) : stream(&s) 
    { if (!(*stream >> value)) stream = 0; } 
    const elemT& operator*() const {return value;} 
    istream_iterator<elemT>& operator++() 
    { if (!(*stream >> value)) stream = 0; 
      return *this; 
    }
};
```

will crash
if out of range
out-of-range iterators

- have been advanced beyond the range of valid elements
  - result of illegal advance operations on legal iterators

- all operations are illegal
  - need not crash, but might exhibit “interesting” behavior

golden rule #5: Never advance an iterator beyond its valid range.

- output stream iterators and inserters have no valid range
  - can be incremented infinitely often

agenda

- valid iterators and iterator ranges
- “invalid” iterators
  - singular iterators
  - past-the-end iterators
  - out-of-range iterators
  - dangling iterators
  - inconsistent iterators
**dangling iterators - definition**

- a dangling iterator points to a sequence element
  - that does not exists or
  - was moved to a different memory location or
  - is otherwise not accessible

- all operations on dangling iterators
  - exhibit undefined behavior

- dangling iterators can inadvertently be created
  - due to lifetime dependencies
  - due to operations that invalidate iterators

**why do we care?**

- lifetime dependencies are frequently overlooked
- invalidation through operations is even less obvious

example: stream iterators depend on the stream

```cpp
istream_iterator<string> in ifstream("in.txt"), eof;
copy(in,eof,ostream_iterator<string>(cout,"\n");
```

problem:
- lifetime of temporary stream object ceases at end of statement ⇒ file closed ⇒ dangling iterator
- possible results: program crash
**recommendation**

Never use temporary stream objects in conjunction with stream iterators.

- A stream iterator is like a pointer to a stream
- Don’t point to anything ephemeral

```cpp
ifstream inFil("in.txt");
istream_iterator<string> in(inFil), eof;
copy(in, eof, ostream_iterator<string>(cout, "\n"));
```

**lifetime dependencies**

- Iterators need a sequence over which they iterate
- The sequence must live longer than the iterator

Examples:

- Container iterator (or pointer to array) needs container (or array) ⇒ Container (or array) must live longer
- Stream iterator needs stream ⇒ Stream must live longer
- Insert iterator needs container and position (i.e. container iterator) ⇒ Container must live longer ⇒ Container iterator must remain valid
- Iterator adapter needs adaptee (i.e. underlying adapted iterator) ⇒ Underlying iterator must live longer
**dangling iterators**

- Iterators are pointer-like objects
  - Introduce the same lifetime dependencies as pointers
  - Sequence must live longer than iterator
- All operations on dangling iterators are illegal
  - Usually (but not always) lead to a program crash

---

**golden rule #7:**

Iterators are “pointers”. Keep an eye on lifetime dependencies between iterator and container.

- Stream iterators depend on stream
- Container iterators depend on container
- Iterator adapters depend on adaptee

---

**agenda**

- Valid iterators and iterator ranges
- “Invalid” iterators
  - Singular iterators
  - Past-the-end iterators
  - Out-of-range iterators
  - Dangling iterators
  - Inconsistent iterators
**inconsistent iterators - definition**

Inconsistent iterators are iterators that return unexpected values when they are dereferenced.

- can happen as a side-effect of `erase()` and `insert()` on `vector` or `deque`
- can be the result of a modifying algorithm

Dereferencing an inconsistent iterator is invalid in the sense that it yields unexpected results.

**inconsistent iterators - examples**

- inconsistent iterator after modifying algorithm:

```cpp
string arr[500];
... fill with elements ... 
string* where = find(arr, arr+500, "Tom");
sort(arr, arr+500);
cout << *where << endl;  // need not print: Tom
```

- inconsistent iterator after `erase()`:

```cpp
vector<string> vec(arr, arr+500);
vector<string>::iterator where
  = find(vec.begin(), vec.end(), "Tom");
vec.erase(vec.begin(), where);
cout << *where << endl;  // need not print: Tom
```
why do we care?

- inconsistent iterators are side effects of operations and algorithms
- occasionally programmers are not aware of the side effects

compare:

```cpp
list<acc> clients(...);
list<acc>::iterator pos = ... position ...
clients.remove_if(inDebt());
cout<<*pos<<endl;
```

to:

```cpp
vector<acc> clients(...);
vector<acc>::iterator pos = ... position ...
remove_if(clients.begin(), clients.end(), inDebt());
cout<<*pos<<endl;
```
\texttt{remove\_if()} on list

- iterator is not affected
  - unless it points to one of the removed elements

\texttt{remove\_if()} on vector

- iterator is affected
  - if it points to a position after the first point of removal
inconsistent iterators

happen as side effect of

- container operations
  - `insert()` and `erase()` on `vector` and `deque`
- algorithms
  - “inplace” algorithms (modify input sequence)
    - `remove()`, `sort()`, `partition()`, `replace()`, ...
  - “copy” algorithms (modify output sequence)
    - `remove_copy()`, `transform()`, `merge()`, ...
- functors
  - functors supplied to algorithms or container operations might modify element content
  - is prohibited, but not enforced

modifying functor - example (prohibited)

- count frequent flyers and raise their status

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

- clearly a modification of sequence elements
  - leads to “inconsistent” iterators
  - prohibited by the standard, but cannot be prevented
inside an algorithm

```cpp
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

modifying functor - example (permitted)

- modification through functor of `for_each`

```cpp
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; }
};

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

- return surprising results on dereferencing
  - side effect of erase() and insert() on vector and deque
  - side effect of modifying algorithms
  - side effect of modifying functors

- all operations are legal
  - but element content is “interesting”

golden rule #8:

Mind modifications of the element content through container operations, algorithms and functors.

agenda

- valid iterators and iterator ranges
- “invalid” iterators
  - singular iterators
  - past-the-end iterators
  - out-of-range iterators
  - dangling iterators
  - inconsistent iterators
- case study
**insertion pitfall**

`template <class Container>`

```cpp
template <class Container> 
void repeatedPrepend(Container src, size_t N) 
{ 
  Container buf; 
  insert_iterator<Container> insIter(buf,buf.begin()); 
  for (int i=0; i<N; i++) 
  { 
    copy(src.begin(),src.end(),insIter); 
  } 
}
```

- results: `(src: A B C, N: 3)`

- vector: A B C crash
- deque: A A B C A B C B C or same as vector
- list: A B C A B C A B C
- multiset: A A A B B B C C C

**VS .NET Connections**

**insertion pitfall**

- every iteration (triggered via the insert iterator) invokes the container’s `insert()` operation
- insertion can invalidate iterators
  - vector: insertion invalidates all iterators after the point of insertion; in case of reallocation invalidates all iterators
  - deque: insertion invalidates all iterators before or after the point of insertion
  - list, (multi)set, (multi)map: insertion does not invalidate any iterators
**insertion into vector**

```cpp
vector<string> buf;
vector<string>::iterator insAt = ... some position ...
buf.insert(insAt, "Don");
```

- insertion into `vector` invalidates positions *after* the point of insertion
  - includes point of insertion

---

**effect of `vector::insert`**

- **start**
  - Amy
  - Anna
  - Bob
  - Bill
- **insAt**
  - Julie
  - Lee
  - Mary
  - Ron
- **finish**
  - endOfStore

- **start**
  - Amy
  - Anna
  - Bob
  - Bill
- **insAt**
  - Julie
  - Lee
  - Mary
  - Ron
- **finish**
  - endOfStore

- **start**
  - Amy
  - Anna
  - Bob
  - Bill
  - Don
- **insAt**
  - Julie
  - Lee
  - Mary
  - Ron
- **finish**
  - endOfStore
- if capacity is exhausted
  - new block of memory is allocated
  - all values are copied and old memory is deleted

⇒ all iterators are invalid
**insertion into vector**

- **start**
  - Amy
  - Anna
  - Bob
  - Bill
  - Marc
  - Nena
  - Dick
  - Tom
  - Don

- **insAt**
  - Julie
  - Lee
  - Mary
  - Ron

- **finish == endOfStore**

---

**dangling vector iterators**

- Reallocation of a vector's internal array invalidates all iterators pointing to the vector.
- Reallocation can be triggered by `insert()` and `reserve()`

---

**golden rule #9:**

Don’t re-use iterators pointing to elements in a vector after any calls to `insert()` or `reserve()`.
similar effects with deque

deque<string> buf;
deque<string>::iterator insAt = ... some position ...
buf.insert(insAt, "Don");

problem:
• insertion into deque invalidates positions before or after the point of insertion
  – may includes point of insertion

© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005, 17:52

VS .NET Connections

typical implementation of deque

© Copyright 1995-2002 by Angelika Langer. All Rights Reserved.
http://www.AngelikaLanger.com
last update: 11/6/2005, 17:52

VS .NET Connections
no problem with list

- insertion into list does not invalidate any iterators

not an issue with set

- insertion into set does not invalidate any iterators
  - similar to list
- insertion ignores position anyway
  - insertion always happens at correct position according to sorting order
  - point of insertion is just a hint
    - tree traversal starts at “hint” position
    - speeds up insertion if elements are inserted in order
insertion and dangling iterators

- insertion can invalidate point of insertion
  - details depend on (implementation of) container
  - problematic with vector and deque
  - not an issue for list, (multi)set, and (multi)map

Don’t re-use iterators used as point-of-insertion (insert()) after any insertion. Use the returned iterator.

golden rule #10:

recommendation

- don’t do this:

```cpp
Container buf;
Container iterator insAt = ... some position ... 
buf.insert(insAt,"Don");
```

- prefer this:

```cpp
Container buf;
Container iterator insAt = ... some position ... 
insAt = buf.insert(insAt,"Don");
```

- insert() returns a valid iterator pointing to the newly inserted element
insertion pitfall

• can we now explain the results of using an inserter?

```cpp
template <class Container>
void repeatedPrepend(Container src, size_t N)
{
    Container buf;
    insert_iterator<Container> insIter(buf, buf.begin());
    for (int i = 0; i < N; i++)
    {
        copy(src.begin(), src.end(), insIter);
    }
}
```

• every loop step uses copy of initial inserter
  – but inserter changes as a side effect of the insertion performed in
    the previous step

example using `vector`

vector is empty

• nothing allocated; all pointers are null pointers
• 1st loop step: `insert()` called repeatedly  down  fine
• 2nd loop step: inserter from before 1st step is used  down  crash

result: A B C crash
example using \texttt{deque}

deque is empty

- memory is allocated, but not used

\begin{itemize}
  \item result: A A B C A B C B C
\end{itemize}

or same as vector

\begin{itemize}
  \item memory is allocated, but not used
\end{itemize}

\begin{itemize}
  \item result: A B C A B C A B C
\end{itemize}

\begin{itemize}
  \item list is empty
    \begin{itemize}
      \item pseudo node represents past-the-end position
      \item point of insertion does no change
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item result: A B C A B C A B C
\end{itemize}
example using `multiset`

- multiset is empty
  - pseudo node represents past-the-end position
  - point of insertion is ignored anyway

- result: `A A A B B C C C`

---

**insertion pitfall - solution**

- how do we avoid the problem?
  - use iterator returned by container member function and algorithm

```cpp
template <class Container>
void repeatedPrepend(Container src, size_t N)
{
    Container buf;
    insert_iterator<Container> insIter(buf, buf.begin());
    for (int i = 0; i < N; i++)
    {
        insIter = copy(src.begin(), src.end(), insIter);
    }
}
```
**insert iterators**

- problem with the insert iterator basically was:
  - same insert iterator was re-used
  - although the underlying iterator had become invalid as a side effect of previous iterations

- “regular” use of insert iterators is safe
  - create insert iterator as temporary object
    - via creator function `inserter()`
  - pass as output iterator to an algorithm

**golden rule #11**: Don’t re-use inserter after the underlying iterator has been invalidated. Create insert iterators as temporaries.

---

**contact info**

Angelika Langer

Training & Mentoring

Object-Oriented Software Development in C++ & Java

http://www.AngelikaLanger.com
Thank you!

- Please drop off your session evaluations in the basket at the back of the room!
- Your comments are greatly appreciated!