Design of a HashTable and its Iterators

Jim Fawcett

CSE687 – Object Oriented Design

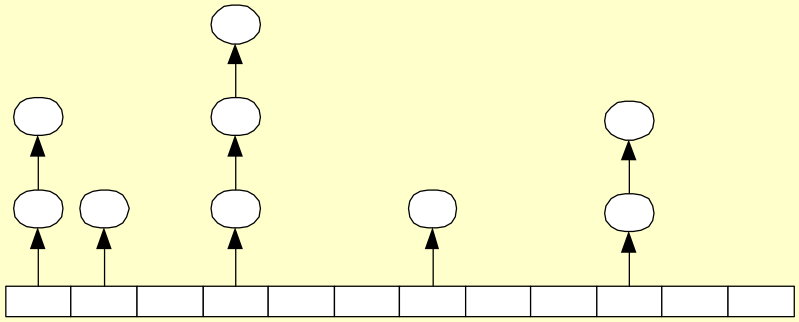
Spring 2007

Iterators as Smart Pointers

- Iterators are "smart" pointers:
  - They provide part, or in some cases, all of the standard pointer interface:

```
*it, it->, ++it, it++, --it, it–, …
```

  - Iterators understand the underlying container structure – often they are friends of the container class:



Containers

First some notes about containers

Containers Often Must Grant Friendship To Their Iterators

```
template < typename key, typename value, typename Hash >
class HashTable
{
  friend class HashIterator;

public:
  typedef key   key_type;
  typedef value value_type;
  typedef HashIterator< key,value,Hash > iterator;
  HashTable(long int size);
  HashTable(const HashTable<key,value,Hash>& ht);
  ~HashTable();

     // lots more stuff here
};
```

Starts with template parameter list.

traits

Does not use template parameters when class name is used as function name.

Uses parameters whenever class name is used as a type.

Destructors Can Get Messy

```
//----< helper function deletes a chain of nodes on heap >-------

template < typename key, typename value, typename Hash >
void HashTable< key,value,Hash >::deleteNodeChain(node<key,value>*
   pNode)
{
  if(pNode == 0) return;
  if(pNode->next() !=0)
    deleteNodeChain(pNode->next());    // recursive call to walk chain
  delete pNode;                        // delete nodes on way back
  pNode = 0;
}
//----< destructor uses helper function >-----------------------

template < typename key, typename value, typename Hash >
HashTable< key,value,Hash >::~HashTable()
{
  for(long int i=0; i<tableSize; ++i)   // delete every chain in table
    deleteNodeChain(table[i]);
}
```

Inserting Nodes into HashTable

```
//----< adds key, value pair to table, returns iterator >--------

template < typename key, typename value, typename Hash >
HashIterator<key,value,Hash>
HashTable< key,value,Hash >::insert(const key& k, const value& v)
{
  unsigned long loc = _hash(k);
  if(Contains(k))   // don't store duplicate keys
  {
    Value(k) = v;   // store value in current node
    return find(k); // return iterator pointing to current n
  }
  ++numElements;    // ok, new key, so add a new node
  node<key,value>* pNode = new node<key,value>(k,v,table[loc]);
  table[loc] = pNode;
  return HashIterator<key,value,Hash>(*this,pNode,loc);
}
```

Stores old pointer to first node in new first nodes' successor pointer.

Puts node at head of linked list of nodes, so new node is pointed to by table cell.

Return iterator pointing to this new node.

Does Container Hold This Key?

```
//----< Contains checks for containment of given key >------

template < typename key, typename value, typename Hash >
bool HashTable< key,value,Hash >::Contains(const key& k) const
{
  unsigned long loc = _hash(k);
  node<key,value>* pNode = table[loc];
  if(pNode == 0) return false;
  do
  {
    if(pNode->Key() == k)
      return true;
    pNode = pNode->next();
  } while(pNode != 0);
  return false;
}
```

Contains(key) is called more often than any other function, so needs to be fast.

- _hash(k) gets to table address quickly.
- Then simple pointer operations

Find Node Containing Key

```
//---< return iterator pointing to node with key >---

template < typename key, typename value, typename Hash >
HashIterator<key,value,Hash>
HashTable< key,value,Hash >::find(const key& k)
{
  unsigned long loc = _hash.operator()(k);
  node<key,value>* pNode = table[loc];
  if(pNode == 0) return end();
  do
  {
    if(pNode->Key() == k)
      return HashIterator<key,value,Hash>(*this,pNode);
    pNode = pNode->next();
  } while(pNode != 0);
  return end();  // return iterator pointing past last element
}
```

Copy Constructor

```
//----< copy constructor>-------------

template < typename key, typename value, typename Hash >
HashTable< key,value,Hash >::HashTable(const HashTable<key,value,Hash>& ht)
         : tableSize(ht.tableSize), _verbose(false)
{
  table = new PointerToNode[tableSize];
  for(long int i=0; i<tableSize; ++i)
    table[i] = 0;
  _hash.Size(tableSize);

  HashIterator<key,value,Hash> it;

  // we know we won't change ht - just reading its values
  // but compiler doesn't know that so we need const_cast

  HashIterator<key,value,Hash> itBeg =
    const_cast< HashTable< key,value,Hash>* >(&ht)->begin();
  HashIterator<key,value,Hash> itEnd =
    const_cast< HashTable< key,value,Hash>* >(&ht)->end();

  it = itBeg;
  while(it != itEnd)
  {
    key k = it->Key();
    value v = it->Value();
    this->insert(k,v);
    ++it;
  }
}
```

Here, we tell HashString function object, _hash, how big the table is. We could not do that with a function. Prefer functors over function pointers!

Create and initial-ize table

it will point to ht

Need to use const_cast to get compiler to let us use iterator on const HashTable<…> ht

Copy in ht's values

On to Iterators

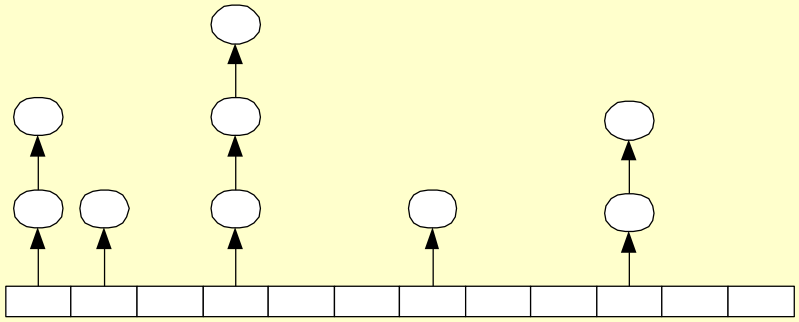
- What is an iterator?
- How do you design an iterator class?
- How do you integrate container and iterator?

Iterators as Smart Pointers

- Iterators are "smart" pointers:
  - They provide part, or in some cases, all of the standard pointer interface:

```
*it, it->, ++it, it++, --it, it–, …
```

  - Iterators understand the underlying container structure – often they are friends of the container class:



```
template <typename key, typename value, typename Hash>
class HashIterator :
  public std::iterator< std::bidirectional_iterator_tag, node<key,value> >
{
public:
  typedef key   key_type;
  typedef value value_type;
  typedef HashIterator<key,value,Hash> iterator;
  HashIterator();
  HashIterator(const HashIterator<key,value,Hash>& hi);
  HashIterator(
    HashTable<key,value,Hash>& ht,
    node<key,value>* pNode = 0,
    long int index=0
  );
  HashIterator<key,value,Hash>&
    operator=(const HashIterator<key,value,Hash>& hi);
  node<key,value>& operator*();
  node<key,value>* operator->();
  iterator& operator++();
  iterator operator++(int);
  iterator& operator--();
  iterator operator--(int);
  bool operator==(const HashIterator<key,value,Hash>& hi) const;
  bool operator!=(const HashIterator<key,value,Hash>& hi) const;
  long int CurrentIndex();

private:
  HashTable<key,value,Hash>* pHashTable;
  node<key,value>* pCurrentNode;
  long int _CurrentIndex;
};
```

traits

Derives from std::iterator

HashIterator can be constructed pointing to node.

Iterator Class

Dereferencing and Selection Operations

```
//----< de-reference operator* >--------------------------

template <typename key, typename value, typename Hash>
node<key,value>& HashIterator<key,value,Hash>::operator*()
{
  return *pCurrentNode;
}

//----< selection operator-> >---------

template <typename key, typename value, typename Hash>
node<key,value>* HashIterator<key,value,Hash>::operator->()
{
  return pCurrentNode;
}
```

Supports the semantics of pointer selection, e.g.:

key_type thisKey = it -> Key();

Returns pointer to node<key,value>. C++ semantics then conspire to select and execute whatever function is selected, in iterator expression, on that node.

Incrementing Operators

```
//----< pre-increment operator++ >---------------
//
//  Return iterator pointing to "next" node.
//  Has to walk both table and node chains.
//
template <typename key, typename value, typename Hash>
HashIterator<key,value,Hash>& HashIterator<key,value,Hash>::operator++()
{
  if(pCurrentNode != 0 && (pCurrentNode = pCurrentNode->next()) != 0)
    return *this;       // next node in chain
  if(_CurrentIndex < pHashTable->tableSize-1)
  {
    long int Index = _CurrentIndex;
    while(pHashTable->table[++Index] == 0)
    {
      if(Index == pHashTable->tableSize-1)
      {
        pCurrentNode = 0;
        ++_CurrentIndex;
        return *this;  // no more nodes
      }
    }
    _CurrentIndex = Index;
    pCurrentNode = pHashTable->table[_CurrentIndex];
    return *this;       // first node in next chain
  }
  pCurrentNode = 0;
  _CurrentIndex = pHashTable->tableSize;
  return *this;
}
```

In node chain, not at end.

Null pointer, so no node chain here.

If we get to end of table, then make iterator = end()

We found a table cell with a node pointer.

Should never reach this point.

Post-Increment Operation

```
//----< post-increment operator++ >-----------------

template <typename key, typename value, typename Hash>
HashIterator<key,value,Hash> HashIterator<key,value,Hash>::operator++(int)
{
  HashIterator<key,value,Hash> temp(*this);  // save current iterator
  operator++();                              // increment internal state
  return temp;                               // return temp of prior state
}
```

Supports the syntax:

```
(it++) -> Value();
```

Post increment and post decrement operators require making temporary iterator objects – what's returned, as well as doing all the work associated with incrementing.

```
//----< default constructor >-----------------------------------

template <typename key, typename value, typename Hash>
HashIterator<key,value,Hash>::HashIterator()
                            : pHashTable(0), pCurrentNode(0) {}

//----< copy constructor >--------------------------------------

template <typename key, typename value, typename Hash>
HashIterator<key,value,Hash>::
HashIterator(const HashIterator<key,value,Hash>& hi)
{
  pHashTable = hi.pHashTable;     // iterator pointing to same table
  pCurrentNode = hi.pCurrentNode;
  _CurrentIndex = hi._CurrentIndex;
}
//----< ctor takes a HashTable, pointer to node, and index >-----
//
//  used only in find(), begin(), and end()
//
template <typename key, typename value, typename Hash>
HashIterator<key,value,Hash>::
HashIterator(
  HashTable<key,value,Hash>& ht,
  node<key,value>* pNode,
  long int index
)
: pHashTable(&ht), pCurrentNode(pNode), _CurrentIndex(index) {}
```

Constructors

End of Containers and Iterators

- Monday's fairly small and simple HashTable has grown some, partly to provide a standard container interface (like the STL containers), but mostly to support iteration over container elements.

- Iterator is best thought of as a smart pointer that knows about the container class and is attached to it through friendship and through many instances of creation by the HashTable container:
  - HashTable<key,value,Hash>::find(key) and insert(key,value) functions return iterators pointing to the found or inserted node.

End of Presentation