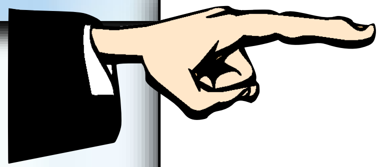
C++ Iterators

ITERATORS

- **An object that traverses a collection of data**
  - During traversal, entries in collection can be modified, added and removed
  - Tracks progress through a collection
  - Knows
    - location in collection
    - if an item has been accessed

```
int listSize = nameList.getLength();
for (int position = 1; position <= listSize; position++)
  std::cout << nameList.getEntry(position) << std::endl;
```



An iterator is an object that traverses a collection of data. During the traversal, you can look at the data entries, modify them, add entries, and remove entries. The Java Class Library contains two interfaces, `Iterator` and `ListIterator`, that specify methods for an iterator. While you could add these iterator methods to the operations of the ADT list, you should instead implement them as a distinct class that interacts with the ADT list. This iterator class can be outside of the ADT list or hidden within its implementation. We will explore both of these approaches in this chapter.

What Is an Iterator?

15.1 How would you count the number of lines on this page? You could use your finger to point to each line as you counted it. Your finger would keep your place on the page. If you paused at a particular line, your finger would be on the current line, and there would be a previous line and a next line. If you think of this page as a list of lines, you would be traversing the list as you counted the lines.

An **iterator** is a program component that enables you to step through, or **traverse**, a collection of data such as a list, beginning with the first entry. During one complete traversal, or **iteration**, each data item is considered once. You control the progress of the iteration by repeatedly asking the iterator to give you a reference to the next entry in the collection. You also can modify the collection as you traverse it by adding, removing, or simply changing entries.

videonote
terators and their use

You are familiar with iteration because you have written loops. For example, if `nameList` is a list of strings, you can write the following `for` loop to display the entire list:

```
int listSize = nameList.getLength();
for (int position = 1; position <= listSize; position++)
   System.out.println(nameList.getEntry(position));
```

Here the loop traverses, or **iterates**, through the entries in the list. Instead of simply displaying each entry, we could do other things to or with it.

15.2 Notice that the previous loop is at the client level, since it uses the ADT operation `getEntry` to access the list. For an array-based implementation of the list, `getEntry` can retrieve the desired array entry directly and quickly. But if a chain of linked nodes represents the list's entries, `getEntry` must move from node to node until it locates the desired one. For example, to retrieve the n^{th} entry in the list, `getEntry` would begin at the first node in the chain and then move to the second node, the third node, and so on until it reached the n^{th} node. At the next repetition of the loop, `getEntry` would retrieve the $n + 1^{st}$ entry in the list by beginning again at the first node in the chain and stepping from node to node until it reached the $n + 1^{st}$ node. This wastes time.

Iteration is such a common operation that we could include it as part of the ADT list. Doing so would enable a more efficient implementation than we were just able to achieve at the client level. Notice that the operation `toArray` of the ADT list performs a traversal. It is an example of a traversal controlled by the ADT. A client can invoke `toArray` but cannot control its traversal once it begins.

But `toArray` only returns the list's entries. What if we want to do something else with them as we traverse them? We do not want to add another operation to the ADT each time we think of another way to use an iteration. We need a way for a client to step through a collection of data and retrieve or modify the entries. The traversal should keep track of its progress; that is, it should know where it is in the collection and whether it has accessed each entry. An iterator provides such a traversal.

Note: **Iterators**
An iterator is a program component that steps through, or traverses, a collection of data. The iterator keeps track of its progress during the traversal, or iteration. It can tell you whether a next entry exists and, if so, return a reference to it. During one cycle of the iteration, each data item is considered once.

ITERATORS

C++ Iterator Operations	Operators
Return the item that the iterator currently references	`*`
Move the iterator to the next item in the list	`++`
Move the iterator to the previous item in the list (used only for bidirectional or random iterators)	`--`
Compare two iterators for equality	`==`
Compare two iterators for inequality	`!=`
Return iterator to first item of container	`begin()`
Return iterator to last item or container	`end()`

Pearson

ITERATORS

- `myList` is a `LinkedList`
  - Each call to `getEntry` requires us to count from the first entry to the entry at `currentPosition`
  - Displaying entire list in this scenario is $O(n^2)$
- **An iterator maintains the current position in the list**
  - Accessing current entry is `O(1)`
  - Process entire list is `O(n)`

```
int currentPosition = 1;
while (currentPosition <= myList.getLength())
{
   std::cout << myList.getEntry(currentPosition); // O(n)
   currentPosition++;
}  // end while
```

```
LinkedIterator<ItemType> currentIterator = myList.begin();
while (currentIterator != myList.end())
{
   std::cout << *currentIterator // O(1) operation
   ++currentIterator;
}  // end while
```

Pearson

ITERATORS

```
template<class ItemType>
class LinkedList;

template <class ItemType>
class LinkedIterator
{
private:
   std::shared_ptr<Node<ItemType>> currentItemPtr;

public:
   typedef std::forward_iterator_tag iterator_category;
   typedef std::ptrdiff_t difference_type;
   typedef ItemType value_type;
   typedef ItemType* pointer;
   typedef ItemType& reference;

   LinkedIterator(std::shared_ptr<Node<ItemType>> nodePtr = nullptr);

   const ItemType operator*();

   LinkedIterator<ItemType> operator++();

   bool operator==(const LinkedIterator<ItemType>& rightHandSide) const;

   bool operator!=(const LinkedIterator<ItemType>& rightHandSide) const;

}; // end LinkedIterator
```

LinkedIterator.h

Input iterator
`input_iterator_tag`
Equality/inequality (`==`, `!=`), access collection entry (`*`)

Output iterator
`output_iterator_tag`
Change a collection entry (`*`)

Forward iterator
`forward_iterator_tag`
Same as the input and output iterators and has a default constructor

Bidirectional iterator
`bidirectional_iterator_tag`
Same as the forward iterator, but also can traverse the collection backward (`--`)

Pearson

ITERATORS

```
template<class ItemType>
class LinkedList;

template <class ItemType>
class LinkedIterator
{
private:
   std::shared_ptr<Node<ItemType>> currentItemPtr;

public:
   typedef std::forward_iterator_tag iterator_category;
   typedef std::ptrdiff_t difference_type;
   typedef ItemType value_type;
   typedef ItemType* pointer;
   typedef ItemType& reference;

   LinkedIterator(std::shared_ptr<Node<ItemType>> nodePtr = nullptr);

   const ItemType operator*();

   LinkedIterator<ItemType> operator++();

   bool operator==(const LinkedIterator<ItemType>& rightHandSide) const;

   bool operator!=(const LinkedIterator<ItemType>& rightHandSide) const;

}; // end LinkedIterator
```

LinkedIterator.h

Random-access iterator

`random_iterator_tag`

Same as the bidirectional iterator and adds support for arithmetic (+, –, +=, –=) and relational (<, <=, >, >=) operations between iterators. Supports the [ ] operator to directly access collection entries.

Pearson

ITERATORS

```
template <class ItemType>
const ItemType LinkedIterator<ItemType>::operator*()
{
   return currentItemPtr->getItem();
}  // end operator*

template <class ItemType>
LinkedIterator<ItemType> LinkedIterator<ItemType>::operator++()
{
   currentItemPtr = currentItemPtr->getNext();
   return *this;
}  // end prefix operator++


template <class ItemType>
bool LinkedIterator<ItemType>::operator==(const
                  LinkedIterator<ItemType>& rightHandSide) const
{
   return (currentItemPtr == rightHandSide.currentItemPtr);
}  // end operator==
```

LinkedIterator.cpp

Pearson

ITERATORS

```
template<class ItemType>
class LinkedList : public ListInterface<ItemType>
{
private:
   std::shared_ptr<Node<ItemType>> headPtr;
   int itemCount;

   auto getNodeAt(int position) const;

public:
   LinkedList();
   LinkedList(const LinkedList<ItemType>& aList);
   virtual ~LinkedList();

   bool isEmpty() const;
   int getLength() const;
   bool insert(int newPosition, const ItemType& newEntry);
   bool remove(int position);
   void clear();
   ItemType getEntry(int position) const throw(PrecondViolatedExcep);
   void setEntry(int position, const ItemType& newEntry)
                        throw(PrecondViolatedExcep);

   LinkedIterator<ItemType> begin();
   LinkedIterator<ItemType> end();
}; // end LinkedList

#include "LinkedList.cpp"
#endif
```

LinkedList.h

m

that access our list.

```
template <class ItemType>
LinkedIterator<ItemType> LinkedList<ItemType>::begin()
{
   return LinkedIterator<ItemType>(headPtr);
} // end begin

template <class ItemType>
LinkedIterator<ItemType> LinkedList<ItemType>::end()
{
   return LinkedIterator<ItemType>(nullptr);
} // end end
```

LinkedList.cpp