Sample Solution to Assignment 3, Problem 1

COURSE HOME
SYLLABUS
CALENDAR
GETTING STARTED
LECTURE NOTES
ASSIGNMENTS
RELATED RESOURCES
DOWNLOAD COURSE MATERIALS

« Back to Assignments

Look in `list.h` for a sense of the structure of the solution. The big idea to speed up the reduce/apply functions while also giving users a nice way to iterate over the items in the list is to create an "iterator" type within our class. Users will be able to write code similar to the STL:

```
// Print out every item in the list
for( List::iterator it = list.begin(); it != list.end(); ++it ) {
    std::cout < < *it << "\n";
}
```

To speed up our "append" function, the List class will also store a pointer to the very last element in the current list.

Directory structure:

- GRADER_INFO.txt
- include
  - apply.h
  - list.h
  - list_node.h
  - reduce.h
- Makefile
- src
  - apply.cpp
  - list.cpp
  - list_iterator.cpp
  - list_node.cpp
  - reduce.cpp
  - test.cpp

*Here are the contents of **apply.h**:*

```
#ifndef _6S096_CPPLIST_APPLY_H
#define _6S096_CPPLIST_APPLY_H
#include "list.h"

class ApplyFunction {
protected:
  virtual int function( int x ) const = 0;
public:
  void apply( List &list ) const;
  virtual ~ApplyFunction() {}
};

// An example ApplyFunction (see apply.cpp)
class SquareApply : public ApplyFunction {
  int function( int x ) const;
};

#endif // _6S096_CPPLIST_APPLY_H
```

*Here are the contents of **list.h**:*

```
#ifndef _6S096_CPPLIST_H
#define _6S096_CPPLIST_H
#include <cstddef>
#include <stdexcept>

class ApplyFunction;
class ReduceFunction;
class ListNode;

class List {
  size_t _length;
  ListNode *_begin;
  ListNode *_back;

public:
  // Can use outside as List::iterator type
  class iterator {
    // Making List a friend class means we'll be able to access
    // the private _node pointer data within the scope of List.
    friend class List;
    ListNode *_node;
```

```
public:
  iterator( ListNode *theNode );
  iterator& operator++();
  int& operator*();
  bool operator==( const iterator &rhs );
  bool operator!=( const iterator &rhs );
};
// Can use outside as List::const_iterator type
class const_iterator {
  // Again, this is basically the only situation you should
  // be using the keyword 'friend'
  friend class List;
  ListNode *_node;
public:
  const_iterator( ListNode *theNode );
  const_iterator& operator++();
  const int& operator*();
  bool operator==( const const_iterator &rhs );
  bool operator!=( const const_iterator &rhs );
};

List();
List( const List &list );
List& operator=( const List &list );
~List();
size_t length()const;
int& value( size_t pos );
int value( size_t pos ) const;
bool empty() const;

iterator begin();
const_iterator begin() const;
iterator back();
const_iterator back() const;
iterator end();
const_iterator end() const;

iterator find( iterator s, iterator t, int needle );
void append( int theValue );
void deleteAll( int theValue );
void insertBefore( int theValue, int before );
void insert( iterator pos, int theValue );
```

```
  void apply( const ApplyFunction &interface );
  int reduce( const ReduceFunction &interface ) const;
  void print() const;
  void clear();

private:
  ListNode* node( iterator it ) { return it._node; }
  ListNode* node( const_iterator it ) { return it._node; }
};

class ListOutOfBounds : public std::range_error {
public:
  explicit ListOutOfBounds() : std::range_error( "List index out of bounds" ) {}
};

#endif // _6S096_CPPLIST_H
```

*Here are the contents of **list_node.h**:*

```
#ifndef _6S096_CPPLIST_NODE_H
#define _6S096_CPPLIST_NODE_H

class ListNode {
  int _value;
  ListNode *_next;
  ListNode( const ListNode & ) = delete;
  ListNode& operator=( const ListNode & ) = delete;
public:
  ListNode();
  ListNode( int theValue );
  ~ListNode();
  int& value();
  int value() const;
  ListNode* next();
  void insertAfter( ListNode *before );
  void setNext( ListNode *nextNode );
  static void deleteNext( ListNode *before );
  static void deleteSection( ListNode *before, ListNode *after );

  static ListNode* create( int theValue = 0 );
};
```

```
#endif // _6S096_CPPLIST_NODE_H
```

*Here are the contents of **reduce.h**:*

```
#ifndef _6S096_CPPLIST_REDUCE_H
#define _6S096_CPPLIST_REDUCE_H
#include "list.h"

class ReduceFunction {
protected:
  virtual int function( int x, int y ) const = 0;
public:
  int reduce( const List &list ) const;
  virtual int identity() const = 0;
  virtual ~ReduceFunction() {}
};

// An example ReduceFunction
class SumReduce : public ReduceFunction {
  int function( int x, int y ) const;
public:
  SumReduce() {}
  ~SumReduce() {}
  int identity() const { return 0; }
};

// Another ReduceFunction
class ProductReduce : public ReduceFunction {
  int function( int x, int y ) const;
public:
  ProductReduce() {}
  ~ProductReduce() {}
  int identity() const { return 1; }
};

#endif // _6S096_CPPLIST_REDUCE_H
```

*Here is the source code file **apply.cpp**:*

```
#include "list.h"
#include "apply.h"

void ApplyFunction::apply( List &list ) const {
```

```
  for( auto it = list.begin(); it != list.end(); ++it ) {
    *it = function( *it );
  }
}

int SquareApply::function( int x ) const {
  return x * x;
}
```

*Here is the source code file **list.cpp**:*

```
#include "list.h"
#include "list_node.h"
#include "apply.h"
#include "reduce.h"

#include <iostream>

List::List() : _length{0}, _begin{ nullptr }, _back{ nullptr } {}

List::List( const List &list ) : _length{0}, _begin{nullptr}, _back{nullptr} {
  for( auto it = list.begin(); it != list.end(); ++it ) {
    append( *it );
  }
}

List& List::operator=( const List &list ) {
  if( this != &list ) {
    clear();
    for( auto it = list.begin(); it != list.end(); ++it ) {
      append( *it );
    }
  }
  return *this;
}

List::~List() { clear(); }

size_t List::length() const { return _length; }

int& List::value( size_t pos ) {
  auto it = begin();
  for( size_t i = 0; i < pos && it != end(); ++it, ++i );
```

```
  if( it == end() ) {
    throw ListOutOfBounds();
  }

  return *it;
}

int List::value( size_t pos ) const {
  auto it = begin();
  for( size_t i = 0; i < pos && it != end(); ++it, ++i );
  if( it == end() ) {
    throw ListOutOfBounds();
  }

  return *it;
}

bool List::empty() const {
 return _length == 0;
}

List::iterator List::begin() { return iterator{ _begin }; }
List::const_iterator List::begin() const { return const_iterator{ _begin }; }
List::iterator List::back() { return iterator{ _back }; }
List::const_iterator List::back() const { return const_iterator{ _back }; }
List::iterator List::end() { return iterator{ nullptr }; }
List::const_iterator List::end() const { return const_iterator{ nullptr }; }

void List::append( int theValue ) {
  auto *newNode = ListNode::create( theValue );

  if( empty() ) {
    newNode->setNext( _back );
    _begin = newNode;
  } else {
    newNode->insertAfter( _back );
  }

  _back = newNode;
  ++_length;
}

void List::deleteAll( int theValue ) {
```

```
  if( !empty() ) {
    // Delete from the front
    while( _begin->value() == theValue && _begin != _back ) {
      auto *newBegin = _begin->next();
      delete _begin;
      _begin = newBegin;
      --_length;
    }

    auto *p = _begin;

    if( _begin != _back ) {
      // Normal deletion from interior of list
      for( ; p->next() != _back; ) {
        if( p->next()->value() == theValue ) {
          ListNode::deleteNext( p );
          --_length;
        } else {
          p = p->next();
        }
      }

      // Deleting the last item
      if( _back->value() == theValue ) {
        ListNode::deleteNext( p );
        _back = p;
        --_length;
      }
    } else if( _begin->value() == theValue ) {
      // Deal with the case where we deleted the whole list
      _begin = _back = nullptr;
      _length = 0;
    }
  }
}

List::iterator List::find( iterator s, iterator t, int needle ) {
  for( auto it = s; it != t; ++it ) {
    if( *it == needle ) {
      return it;
    }
  }
  return t;
```

```
}

void List::insert( iterator pos, int theValue ) {
  auto *posPtr = node( pos );
  auto *newNode = ListNode::create( theValue );
  newNode->insertAfter( posPtr );
  ++_length;
}

void List::insertBefore( int theValue, int before ) {
  if( !empty() ) {
    if( _begin->value() == before ) {
      auto *newNode = ListNode::create( theValue );
      newNode->setNext( _begin );
      _begin = newNode;
      ++_length;
    } else {
      auto *p = _begin;
      for( ; p != _back && p->next()->value() != before; p = p->next() );
      if( p != _back && p->next()->value() == before ) {
        auto *newNode = ListNode::create( theValue );
        newNode->insertAfter( p );
        ++_length;
      }
    }
  }
}

void List::apply( const ApplyFunction &interface ) {
  interface.apply( *this );
}

int List::reduce( const ReduceFunction &interface ) const {
  return interface.reduce( *this );
}

void List::print() const {
  std::cout << "{ ";
  for( auto it = begin(); it != back(); ++it ) {
    std::cout << *it << " -> ";
  }
  if( !empty() ) {
    std::cout << *back() << " ";
```

```
  }
  std::cout << "}\n";
}

void List::clear() {
  for( auto *p = _begin; p != nullptr; ) {
    auto *p_next = p->next();
    delete p;
    p = p_next;
  }
  _length = 0;
  _begin = nullptr;
  _back = nullptr;
}
```

*Here is the source code file **list_iterator.cpp**:*

```
#include "list.h"
#include "list_node.h"

List::iterator::iterator( ListNode *theNode ) : _node{theNode} {}
List::iterator& List::iterator::operator++() {
  _node = _node->next();
  return *this;
}
int& List::iterator::operator*() { return _node->value(); }
bool List::iterator::operator==( const iterator &rhs ) { return _node == rhs._node; }
bool List::iterator::operator!=( const iterator &rhs ) { return _node != rhs._node; }

List::const_iterator::const_iterator( ListNode *theNode ) : _node{theNode} {}
List::const_iterator& List::const_iterator::operator++() {
  _node = _node->next();
  return *this;
}
const int& List::const_iterator::operator*() { return _node->value(); }
bool List::const_iterator::operator==( const const_iterator &rhs ) { return _node == rhs._node; }
bool List::const_iterator::operator!=( const const_iterator &rhs ) { return _node != rhs._node; }
```

*Here is the source code file **list_node.cpp**:*

```
#include "list_node.h"

ListNode::ListNode() : _value{0}, _next{nullptr} {}
ListNode::ListNode( int theValue ) : _value{theValue}, _next{nullptr} {}
```

```
ListNode::~ListNode() {}
int& ListNode::value() { return _value; }
int ListNode::value(){const { return _value; }
ListNode* ListNode::next() { return _next; }

void ListNode::insertAfter( ListNode *before ) {
  _next = before->next();
  before->_next = this;
}

void ListNode::setNext( ListNode *nextNode ) {
  _next = nextNode;
}

void ListNode::deleteNext( ListNode *before ) {
  auto *after = before->next()->next();
  delete before->next();
  before->_next = after;
}

void ListNode::deleteSection( ListNode *before, ListNode *after ) {
  auto *deleteFront = before->next();
  while( deleteFront != after ) {
    auto *nextDelete = deleteFront->next();
    delete deleteFront;
    deleteFront = nextDelete;
  }
}

ListNode* ListNode::create( int theValue ) {
  return new ListNode{ theValue };
}
```

*Here is the source code file **reduce.cpp**:*

```
#include "list.h"
#include "reduce.h"

int ReduceFunction::reduce(const List &list ) const {
  int result = identity();
  for( auto it = list.begin(); it != list.end(); ++it ) {
    result = function( result, *it );
  }
```

```
  return result;
}

int SumReduce::function( int x, int y ) const {
  return x + y;
}

int ProductReduce::function(int x, int y ) const {
  return x * y;
}
```

Below is the output using the test data:

```
cpplist:
 1: OK [0.004 seconds] OK!
 2: OK [0.005 seconds] OK!
 3: OK [0.005 seconds] OK!
 4: OK [0.009 seconds] OK!
 5: OK [0.006 seconds] OK!
 6: OK [0.308 seconds] OK!
 7: OK [0.053 seconds] OK!
 8: OK [0.007 seconds] OK!
 9: OK [0.005 seconds] OK!
10: OK [0.742 seconds] OK!
```

« Back to Assignments