# Project 1 Deliverable

Paul Nickerson

November 24, 2014

# CDAL Informal Documentation

Paul Nickerson

## Something here

this is a test hello world

## Something here

SSLL

# SSLL Informal Documentation

Paul Nickerson

## List Methods

### iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

### iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

### const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

### const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
  - That is, if the list size is zero, then end() == begin()

### T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### SSLL(const SSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

### SSLL& operator=(const SSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

### T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

### void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1

- May be called with a position one past the last stored item, in which case the new item becomes the last
    - In this case we pass the element to push_back(), which can do O(1) insert
    - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

## void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

## T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

## T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

## T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## bool is_empty() const

- Returns true IIF size() == 0

## size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

## void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

## bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list
- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

4

## std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

# Iterator Methods

## explicit SSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

## SSLL_Iter(const SSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

## reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change

## pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

## self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

## Const Iterator Methods

### explicit SSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

### SSLL_Const_Iter(const SSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

### reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile

### pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

### self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

# CDAL Informal Documentation

Paul Nickerson

## Something here

this is a test hello world

## Something here

**PSLL**

# PSLL Informal Documentation

Paul Nickerson

## List Methods

### iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.

### iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

### const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.

### const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

### T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## PSLL()

- Default constructor - initializes the head, tail, and free-head dummy nodes

## PSLL(const PSLL& src)

- Copy constructor - starting from uninitialized state, initialize the class, then use an iterator to push_bash() each source item into the current list
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

## PSLL& operator=(const PSLL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state, 2) initialize the class, and 3) use an iterator to push_bash() each source item into the current list
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

## T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

### void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
    - In this case we pass the element to push_back(), which can do O(1) insert
    - For position < size(), we do a O(N) traversal to the specified position
- Providing a position greater than the current list size should throw an out-of-range error
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown

### void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

### void push_back(const T& element)

- Inserts a new item to the back of the list by converting the current tail to a non-dummy node containing the item and adds a new tail
- If a new node cannot be procured due to memory constraints, an error message is outputted to stderr and std::bad_alloc is thrown
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

### T pop_front()

- Removes the node at head->next and returns its stored item
- Points head->next to the node which the removed node pointed to
- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

### T pop_back()

- Removes the node at position (size() - 1), returning its stored item
- Points preceding_node->next to the tail

- Decrements the list size
- If the list is empty then throw an out-of-range error
- It would be a runtime_error if, after checking that the list is non-empty and prior to popping, head->next == tail. This would indicate internal list state corruption.

## T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left."
- May only be called with positions *less than* the current list size
- It would be a runtime_error if, after checking that the list is non-empty and prior to removing, head->next == tail. This would indicate internal list state corruption.

## T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## bool is_empty() const

- Returns true IIF size() == 0

## size_t size() const

- Returns value of the counter which tracks the number of items stored in the array
- If the item quantity counter is zero, then head->next should == tail. If not, an error should be thrown indicating corrupt internal state
- If the item quantity counter is nonzero, then head->next should != tail. If not, an error should be thrown indicating corrupt internal state

## void clear()

- Removes all elements in the list by calling pop_front() until is_empty() returns true

## bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list

- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

## std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

# Iterator Methods

## explicit PSLL_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

## PSLL_Iter(const PSLL_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

## reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change

## pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

### self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

## Const Iterator Methods

### explicit PSLL_Const_Iter(Node* start)

- Explicit constructor for an iterator which, when dereferenced, will return an immutable reference to the item held at start
- start can be tail, which signals that the iterator points to the end of the list
- start *cannot* be null, otherwise throw a runtime_error because, since only the current class can call this constructor (Node is private), start==nullptr indicates internal state corruption

6

## PSLL_Const_Iter(const PSLL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

## reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile

## pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

## self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

## self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie current_node->is_dummy==true

## self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

## bool operator==(const self_type& rhs) const

- Returns true IIF the currently-held node pointer is the same as rhs's, otherwise returns false

## bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

# CDAL Informal Documentation

Paul Nickerson

## Something here

this is a test hello world

## Something here

**SDAL**

# SDAL Informal Documentation

## Paul Nickerson

## List Methods

### iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.
- Passes a pointer to the end slot so that the iterator can do bounds checking

### iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

### const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.
- Passes a pointer to the end slot so that the iterator can do bounds checking

### const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

1

### T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### SDAL(const SDAL& src)

- Copy constructor - starting from uninitialized state, initialize the class by allocating a number of nodes equal to the source instance's array size, then use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

### SDAL& operator=(const SDAL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninitialized state by freeing the item array, 2) initialize the class by allocating a number of nodes equal to the source instance's array size, and 3) use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

### void embiggen_if_necessary()

- Called whenever we attempt to increase the list size
- Checks if backing array is full, and if so, allocate a new array 150% the size of the original, copy the items over to the new array, and deallocate the original one.
- If we fail to allocate nodes, throw a bad_alloc exception

### void shrink_if_necessary()

- Called whenever we attempt to decrease the list size

- Because we don't want the list to waste too much memory, whenever the array's size is >= 100 slots and fewer than half the slots are used, allocate a new array 50% the size of the original, copy the items over to the new array, and deallocate the original one.
- If we fail to allocate nodes, throw a bad_alloc exception

## T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

## void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- Calls embiggen_if_necessary() to ensure we have space to insert the new item
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
- Providing a position greater than the current list size should throw an out-of-range error

## void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## void push_back(const T& element)

- Inserts a new item to the back of the list calling insert() with the position defined as one past the last stored item
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## T pop_front()

- Wrapper for remove(0)
- Removes the node at item_array[0] and returns its stored item

- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

## T pop_back()

- Wrapper for remove(size() - 1)
- Removes last stored node, returning its item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

## T remove(size_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left" by traversing from the specified slot to the end of the array and moving each item to its preceding slot
- May only be called with positions *less than* the current list size
- It would be an error if, after removing, size() returned anything besides the old value returned from size() minus one

## T item_at(size_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## bool is_empty() const

- Returns true IIF size() == 0

## size_t size() const

- Returns value of the counter which tracks the number of items stored in the array

## void clear()

- Removes all elements in the list by setting the counter holding the list size to zero. No further action is taken as it is assumed that the embiggen/shrink methods will handle it

## bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list
- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

## std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

# Iterator Methods

## explicit SDAL_Iter(T* item_array, T* end_ptr)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the first item held in the item_array parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

## SDAL_Iter(const SDAL_Iter& src)

- Copy constructor - sets the current iterator position in the item array and the end position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

## reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change

### pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

### self_reference operator=(const self_type& src)

- Changes the current and end iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie iter==iter_end

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

## Const Iterator Methods

### explicit SDAL_Const_Iter(Node* start)

- Explicit constructor for an iterator which returns an immutable reference to the first item held in the item_array parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

## SDAL_Const_Iter(const SDAL_Const_Iter& src)

- Copy constructor - sets the iterator's current position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

## reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile

## pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

## self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

## self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie iter==iter_end

## self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

## bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

## bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

# CDAL Informal Documentation

Paul Nickerson

## Something here

this is a test hello world

## Something here

**CDAL**

# CDAL Informal Documentation

Paul Nickerson

## List Methods

### iterator begin()

- Creates an iterator which, when dereferenced, returns a mutable reference to the first stored item.
- Passes a pointer to the end slot so that the iterator can do bounds checking

### iterator end()

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

### const_iterator begin() const

- Creates an iterator which, when dereferenced, returns an immutable reference to the first stored item.
- Passes a pointer to the end slot so that the iterator can do bounds checking

### const_iterator end() const

- Creates an iterator corresponding to the slot one past the end of the list.
- Used in conjunction with const_iterator begin() to traverse every item in the list.
- Dereferencing the resulting iterator should throw an error.
- The iterator returned is "list size" increment operations past the incrementor returned by begin()
    - That is, if the list size is zero, then end() == begin()

1

### T& operator

- Returns a mutable reference to the item at position i, so when the resulting reference is changed, the item should update in the list as well
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

### const T& operator const

- Returns an immutable reference to the item at position i, so that the reference cannot be used to change the list's copy of the item
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## CDAL()

- Default constructor - initializes the class by allocating head/tail dummy nodes, then adding an initial node

## CDAL(const CDAL& src)

- Copy constructor - starting from uninitialized state, initialize the class by allocating head/tail dummy nodes, then use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

## CDAL& operator=(const CDAL& src)

- Copy assignment operator - starting from an arbitrary state, 1) reset to uninialized state by freeing all the items, 2) initialize the class by allocating a number of nodes equal to the source instance's array size, and 3) use an iterator to push_bash() each source item into the current list
- If we fail to allocate nodes, throw a bad_alloc exception
- Returns a reference to *this, the copied-to instance
- Afterwards, this->size() should equal src.size(). If not, throw a runtime_error

## void embiggen_if_necessary()

- Called whenever we attempt to increase the list size
- If each array slot in every link is filled and we want to add a new item, allocate and append a new link by transforming the tail node into a usable item array container that points to a freshly-allocated tail node
- If we fail to allocate nodes, throw a bad_alloc exception

## void shrink_if_necessary()

- Called whenever we attempt to decrease the list size
- Because we don't want the list to waste too much memory, whenever more than half of the arrays are unused (they would all be at the end of the chain), we deallocate half the arrays by traversing to the last node to keep, then dropping each subsequent node until we reach the tail

## T replace(const T& element, size_t position)

- Replaces the currently-stored element at the specified position with a copy of the specified element
- Returns a copy of the item that was stored at the specified position
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error
- The size() of the list should remain unchanged before and after

## void insert(const T& element, size_t position)

- Inserts a copy of the specified element to the list at the specified position, shifting the element originally at that and those in subsequent positions one position to the "right."
- Calls embiggen_if_necessary() to ensure we have space to insert the new item
- List size gets incremented by 1
- May be called with a position one past the last stored item, in which case the new item becomes the last
- Providing a position greater than the current list size should throw an out-of-range error

## void push_front(const T& element)

- Inserts a new item to the front of the list by calling insert(element, 0), incrementing the list size by one
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## void push_back(const T& element)

- Inserts a new item to the back of the list calling insert() with the position defined as one past the last stored item
- It would be an error if, after pushing, size() returned anything besides one plus the old value returned from size()

## T pop\_front()

- Wrapper for remove(0)
- Removes the node at item\_array[0] and returns its stored item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

## T pop\_back()

- Wrapper for remove(size() - 1)
- Removes last stored node, returning its item
- If the list is empty then throw an out-of-range error
- It would be an error if, after popping, size() returned anything besides the old value returned from size() minus one

## T remove(size\_t position)

- Removes and returns the the element at the specified position, shifting the subsequent elements one position to the "left" by traversing from the specified slot in the node's array to the end of the last node's item array and moving each item to its preceding slot
- May only be called with positions *less than* the current list size
- It would be an error if, after removing, size() returned anything besides the old value returned from size() minus one

## T item\_at(size\_t position) const

- A wrapper for operator[] which return a copy of the item at position i, so when the resulting reference is changed, the item should not update in the list
- Providing a position greater than *or equal to* the current list size should throw an out-of-range error

## bool is\_empty() const

- Returns true IIF size() == 0

## size\_t size() const

- Returns value of the counter which tracks the number of items stored in the array

## void clear()

- Removes all elements in the list by setting the counter holding the list size to zero. No further action is taken as it is assumed that the embiggen/shrink methods will handle it

## bool contains(const T& element, bool equals(const T& a, const T& b)) const

- Returns true IFF one of the elements of the list matches the specified element.
- Uses an iterator to traverse the list
- At each position, calls the equals callback function. If that returns true, stop iterating and return true
- If the end position is reached before the item is found, return false
- It would be a runtime_error if an item was inserted and calling contains() with that item returned false, which would indicate internal state corruption
- It would be a runtime_error if an item existed in one list and then, after making a copy of that list, the copy did not contain the item (internal state corruption)

## std::ostream& print(std::ostream& out) const

- Passes a string of the form [item1,item2,item3] to the provided output stream
- If the list contains no items, passes to the output stream
- It would be an error if print() yielded different results from two lists which should be the same (eg constructed the same, copied, assigned, etc)

# Iterator Methods

## CDAL_Iter(ItemLoc const& here)

- Explicit constructor for an iterator which, when dereferenced, will return a mutable reference to the item held at the node and array index described by the here parameter
- Neither item_array nor end_ptr may be null
- end_ptr must be greater than or equal to item_array

## CDAL_Iter(const CDAL_Iter& src)

- Copy constructor - sets the current iterator position to the node and array index described by src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

### reference operator*() const

- Returns a mutable reference to the item held at the current iterator position
- It would be an error if the client properly attempted to change the value of the returned reference and the stored item value did not change

### pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()

### self_reference operator=(const self_type& src)

- Changes the current and end iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie curr_node->is_dummy

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

## Const Iterator Methods

### CDAL_Iter(ItemLoc const& here)

- Explicit constructor for an iterator which, when dereferenced, returns an immutable reference to the item held at the node and array index described by the here parameter

### CDAL_Const_Iter(const CDAL_Const_Iter& src)

- Copy constructor - sets the current iterator position to the node and array index described by src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption

### reference operator*() const

- Returns an immutable reference to the item held at the current iterator position
- The const keyword in the reference typedef guarantees that code which attempts to modify the referenced item will not compile

### pointer operator->() const

- Returns a pointer to the item held at the current iterator position by returning the value of operator*() with the address-of operator applied
- The same validation measures apply here as to operator*()
- The const keyword in the pointer typedef guarantees that code which attempts to modify the referenced item will not compile

### self_reference operator=(const self_type& src)

- Changes the current iterator position to that of src
- Afterwards, operator==(src) should return true, otherwise throw a runtime_error indicating state corruption
- Returns a reference to current instance

### self_reference operator++()

- Prefix increment operator - increments the current iterator then returns it as a reference
- Should throw an out-of-range error if we're at the end of the list, ie curr_node->is_dummy

### self_type operator++(int)

- Postfix increment operator - creates a pre-incremented copy of the current instance, increments the current iterator (calls prefix operator directly, so its sanity checks apply to this method), then returns the copied instance

### bool operator==(const self_type& rhs) const

- Returns true IIF the current and end iter pointers match between current instance and rhs, otherwise returns false

### bool operator!=(const self_type& rhs) const

- Returns true IIF operator==() returns false, otherwise returns trus

# CDAL Informal Documentation

Paul Nickerson

## Something here

this is a test hello world

## Something here

**SSLL checklist & source code**

# ssll/checklist.txt

Simple, Singly Linked List written by Nickerson, Paul
COP 3530, 2014F 1087
======================================================================
Part I:
======================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

My LIST implementation 100% correctly supports the following methods
as described in part I:

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes


======================================================================
Part II:
======================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following

methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

=======================================================================

Part III:
========================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My LIST implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Simple, Singly Linked List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

======================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
======================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt

# ssll/source/SSLL.h

```
1  //note to self: global search for todo and xxx before turning this assignment in
2
3
4
5
6
7
8
9
10
11
12
13
14  #ifndef _SSLL_H_
15  #define _SSLL_H_
16
17  // SSLL.H
18  //
19  // Singly-linked list (non-polymorphic)
20  //
21  // Authors: Paul Nickerson, Dave Small
22  // for COP 3530
23  // 201409.16 - created
24
25  #include <iostream>
26  #include <stdexcept>
27  #include <cassert>
28
29  namespace cop3530 {
30      template <class T>
31      class SSLL {
32      private:
33          struct Node {
34              T item;
35              Node* next;
36              bool is_dummy;
37          }; // end struct Node
38          size_t num_items;
39          Node* head;
40          Node* tail;
41          Node* node_at(size_t position) const {
42              Node* n = head->next;
43              for (size_t i = 0; i != position; ++i, n = n->next);
44              return n;
45          }
46          Node* node_before(size_t position) const {
47              if (position == 0)
```

```
48              return head;
49          else
50              return node_at(position - 1);
51      }
52      Node* allocate_new_node() {
53          Node* n;
54          try {
55              n = new Node();
56          } catch (std::bad_alloc& ba) {
57              std::cerr << "allocate_new_node(): failed to allocate memory for new
                      node" << std::endl;
58              throw std::bad_alloc();
59          }
60          return n;
61      }
62      Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
           false) {
63          Node* n = allocate_new_node();
64          n->is_dummy = dummy;
65          n->item = element;
66          n->next = next;
67          return n;
68      }
69      Node* design_new_node(Node* next = nullptr, bool dummy = false) {
70          Node* n = allocate_new_node();
71          n->is_dummy = dummy;
72          n->next = next;
73          return n;
74      }
75      void init() {
76          num_items = 0;
77          try {
78              tail = design_new_node(nullptr, true);
79              head = design_new_node(tail, true);
80          } catch (std::bad_alloc& ba) {
81              std::cerr << "init(): failed to allocate memory for head/tail nodes"
                      << std::endl;
82              throw std::bad_alloc();
83          }
84      }
85      //note to self: the key to simple ssll navigation is to frame the problem
           in terms of the following two functions (insert_node_after and
           remove_item_after)
86      void insert_node_after(Node* existing_node, Node* new_node) {
87          existing_node->next = new_node;
88          ++num_items;
89      }
90      //destroys the subsequent node and returns its item
91      T remove_item_after(Node* preceeding_node) {
92          Node* removed_node = preceeding_node->next;
93          T item = removed_node->item;
94          preceeding_node->next = removed_node->next;
```

```
 95              delete removed_node;
 96              --num_items;
 97              return item;
 98          }
 99          void copy_constructor(const SSLL& src) {
100              const_iterator fin = src.end();
101              for (const_iterator iter = src.begin(); iter != fin; ++iter) {
102                  push_back(*iter);
103              }
104              if ( ! src.size() == size())
105                  throw std::runtime_error("copy_constructor: Copying failed - sizes
                         don't match up");
106          }
107      public:
108
109          //----------------------------------------------------
110          // iterators
111          //----------------------------------------------------
112          class SSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
113          {
114          public:
115              // inheriting from std::iterator<std::forward_iterator_tag, T>
116              // automagically sets up these typedefs...
117              typedef T value_type;
118              typedef std::ptrdiff_t difference_type;
119              typedef T& reference;
120              typedef T* pointer;
121              typedef std::forward_iterator_tag iterator_category;
122
123              // but not these typedefs...
124              typedef SSLL_Iter self_type;
125              typedef SSLL_Iter& self_reference;
126
127          private:
128              Node* here;
129
130          public:
131              explicit SSLL_Iter(Node* start) : here(start) {
132                  if (start == nullptr)
133                      throw std::runtime_error("SSLL_Iter: start cannot be null");
134              }
135              SSLL_Iter(const SSLL_Iter& src) : here(src.here) {
136                  if (*this != src)
137                      throw std::runtime_error("SSLL_Iter: copy constructor failed");
138              }
139              reference operator*() const {
140                  return here->item;
141              }
142              pointer operator->() const {
143                  return & this->operator*();
144              }
145              self_reference operator=( const self_type& src ) {
```

```
146            if (&src == this)
147                return *this;
148            here = src.here;
149            if (*this != src)
150                throw std::runtime_error("SSLL_Iter: copy assignment failed");
151            return *this;
152        }
153        self_reference operator++() { // preincrement
154            if (here->is_dummy)
155                throw std::out_of_range("SSLL_Iter: Can't traverse past the end
                        of the list");
156            here = here->next;
157            return *this;
158        }
159        self_type operator++(int) { // postincrement
160            self_type t(*this); //save state
161            operator++(); //apply increment
162            return t; //return state held before increment
163        }
164        bool operator==(const self_type& rhs) const {
165            return rhs.here == here;
166        }
167        bool operator!=(const self_type& rhs) const {
168            return ! operator==(rhs);
169        }
170    };
171
172    class SSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
173    {
174    public:
175        // inheriting from std::iterator<std::forward_iterator_tag, T>
176        // automagically sets up these typedefs...
177        typedef T value_type;
178        typedef std::ptrdiff_t difference_type;
179        typedef const T& reference;
180        typedef const T* pointer;
181        typedef std::forward_iterator_tag iterator_category;
182
183        // but not these typedefs...
184        typedef SSLL_Const_Iter self_type;
185        typedef SSLL_Const_Iter& self_reference;
186
187    private:
188        const Node* here;
189
190    public:
191        explicit SSLL_Const_Iter(Node* start) : here(start) {
192            if (start == nullptr)
193                throw std::runtime_error("SSLL_Const_Iter: start cannot be null");
194        }
195        SSLL_Const_Iter(const SSLL_Const_Iter& src) : here(src.here) {
196            if (*this != src)
```

```cpp
197                    throw std::runtime_error("SSLL_Const_Iter: copy constructor
                           failed");
198            }
199
200            reference operator*() const {
201                return here->item;
202            }
203            pointer operator->() const {
204                return & this->operator*();
205            }
206            self_reference operator=( const self_type& src ) {
207                if (&src == this)
208                    return *this;
209                here = src.here;
210                if (*this != src)
211                    throw std::runtime_error("SSLL_Const_Iter: copy assignment
                               failed");
212                return *this;
213            }
214            self_reference operator++() { // preincrement
215                if (here->is_dummy)
216                    throw std::out_of_range("SSLL_Const_Iter: Can't traverse past the
                               end of the list");
217                here = here->next;
218                return *this;
219            }
220            self_type operator++(int) { // postincrement
221                self_type t(*this); //save state
222                operator++(); //apply increment
223                return t; //return state held before increment
224            }
225            bool operator==(const self_type& rhs) const {
226                return rhs.here == here;
227            }
228            bool operator!=(const self_type& rhs) const {
229                return ! operator==(rhs);
230            }
231        };
232
233        //---------------------------------------------------
234        // types
235        //---------------------------------------------------
236        typedef T value_type;
237        typedef SSLL_Iter iterator;
238        typedef SSLL_Const_Iter const_iterator;
239
240        iterator begin() { return SSLL_Iter(head->next); }
241        iterator end() { return SSLL_Iter(tail); }
242
243        const_iterator begin() const { return SSLL_Const_Iter(head->next); }
244        const_iterator end() const { return SSLL_Const_Iter(tail); }
245
```

```cpp
        //------------------------------------------------
        // operators
        //------------------------------------------------
        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        //------------------------------------------------
        // Constructors/destructor/assignment operator
        //------------------------------------------------

        SSLL() {
            init();
        }
        //------------------------------------------------
        //copy constructor
        //note to self: src must be const in case we want to assign this from a
             const source
        SSLL(const SSLL& src) {
            init();
            copy_constructor(src);
        }

        //------------------------------------------------
        //destructor
        ~SSLL() {
            // safely dispose of this SSLL's contents
            clear();
        }

        //------------------------------------------------
        //copy assignment constructor
        SSLL& operator=(const SSLL& src) {
            if (&src == this) // check for self-assignment
                return *this;    // do nothing
            // safely dispose of this SSLL's contents
            clear();
            // populate this SSLL with copies of the other SSLL's contents
            copy_constructor(src);
            return *this;
```

53

```
295            }
296
297            //--------------------------------------------------
298            // member functions
299            //--------------------------------------------------
300
301            /*
302                replaces the existing element at the specified position with the
                        specified element and
303                returns the original element.
304            */
305            T replace(const T& element, size_t position) {
306                T old_item;
307                if (position >= size()) {
308                    throw std::out_of_range(std::string("replace: No element at position
                            ") + std::to_string(position));
309                } else {
310                    //we are guaranteed to be at a non-dummy item now because of the
                            above if statement
311                    Node* iter = node_at(position);
312                    old_item = iter->item;
313                    iter->item = element;
314                }
315                return old_item;
316            }
317
318            //--------------------------------------------------
319            /*
320                adds the specified element to the list at the specified position,
                        shifting the element
321                originally at that and those in subsequent positions one position to the
                        right.
322            */
323            void insert(const T& element, size_t position) {
324                if (position > size()) {
325                    throw std::out_of_range(std::string("insert: Position is outside of
                            the list: ") + std::to_string(position));
326                } else if (position == size()) {
327                    //special O(1) case
328                    push_back(element);
329                } else {
330                    //node_before_position is guaranteed to point to a valid node
                            because we use a dummy head node
331                    Node* node_before_position = node_before(position);
332                    Node* node_at_position = node_before_position->next;
333                    Node* new_node;
334                    try {
335                        new_node = design_new_node(element, node_at_position);
336                    } catch (std::bad_alloc& ba) {
337                        std::cerr << "insert(): failed to allocate memory for new node"
                                << std::endl;
338                        throw std::bad_alloc();
```

```
                }
                insert_node_after(node_before_position, new_node);
            }
        }

        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }

        //-------------------------------------------------
        /*
            appends the specified element to the list.
        */
        void push_back(const T& element) {
            Node* new_tail;
            try {
                new_tail = design_new_node(nullptr, true);
            } catch (std::bad_alloc& ba) {
                std::cerr << "push_back(): failed to allocate memory for new tail"
                    << std::endl;
                throw std::bad_alloc();
            }
            insert_node_after(tail, new_tail);
            //transform the current tail node from a dummy to a real node holding
                element
            tail->is_dummy = false;
            tail->item = element;
            tail->next = new_tail;
            tail = tail->next;
        }

        /*
            removes and returns the element at the list's head.
        */
        T pop_front() {
            if (is_empty()) {
                throw std::out_of_range("pop_front: Can't pop: list is empty");
            }
            if (head->next == tail) {
                throw std::runtime_error("pop_front: head->next == tail, but list
                    says it's not empty (corrupt state)");
            }
            return remove_item_after(head);
        }

        //-------------------------------------------------
        /*
            removes and returns the element at the list's tail.
        */
```

```
388        T pop_back() {
389            if (is_empty()) {
390                throw std::out_of_range("pop_back: Can't pop: list is empty");
391            }
392            if (head->next == tail) {
393                throw std::runtime_error("pop_back: head->next == tail, but list
                       says it's not empty (corrupt state)");
394            }
395            //XXX this is O(N), a disadvantage of this architecture
396            Node* node_before_last = node_before(size() - 1);
397            T item = remove_item_after(node_before_last);
398            return item;
399        }

401        //--------------------------------------------------
402        /*
403            removes and returns the the element at the specified position,
404            shifting the subsequent elements one position to the left.
405        */
406        T remove(size_t position) {
407            T item;
408            if (position >= size()) {
409                throw std::out_of_range(std::string("remove: No element at position
                       ") + std::to_string(position));
410            }
411            if (head->next == tail) {
412                throw std::runtime_error("remove: head->next == tail, but list says
                       it's not empty (corrupt state)");
413            }
414            //using a dummy head node guarantees that there be a node immediately
                   preceeding the specified position
415            Node *node_before_position = node_before(position);
416            item = remove_item_after(node_before_position);
417            return item;
418        }

420        //--------------------------------------------------
421        /*
422            returns (without removing from the list) the element at the specified
                   position.
423        */
424        T item_at(size_t position) const {
425            if (position >= size()) {
426                throw std::out_of_range(std::string("item_at: No element at position
                       ") + std::to_string(position));
427            }
428            return operator[](position);
429        }

431        //--------------------------------------------------
432        /*
433            returns true IFF the list contains no elements.
```

```
434          */
435          bool is_empty() const {
436              return size() == 0;
437          }
438
439          //--------------------------------------------------
440          /*
441              returns the number of elements in the list.
442          */
443          size_t size() const {
444              if (num_items == 0 && head->next != tail) {
445                  throw std::runtime_error("size: head->next != tail, but list says
                        it's empty (corrupt state)");
446              } else if (num_items > 0 && head->next == tail) {
447                  throw std::runtime_error("size: head->next == tail, but list says
                        it's not empty (corrupt state)");
448              }
449              return num_items;
450          }
451
452          //--------------------------------------------------
453          /*
454              removes all elements from the list.
455          */
456          void clear() {
457              while ( ! is_empty()) {
458                  pop_front();
459              }
460          }
461
462          //--------------------------------------------------
463          /*
464              returns true IFF one of the elements of the list matches the specified
                  element.
465          */
466          bool contains(const T& element,
467                  bool equals(const T& a, const T& b)) const {
468              bool element_in_list = false;
469              const_iterator fin = end();
470              for (const_iterator iter = begin(); iter != fin; ++iter) {
471                  if (equals(*iter, element)) {
472                      element_in_list = true;
473                      break;
474                  }
475              }
476              return element_in_list;
477          }
478
479          //--------------------------------------------------
480          /*
481              If the list is empty, inserts "<empty list>" into the ostream;
482              otherwise, inserts, enclosed in square brackets, the list's elements,
```

```cpp
483              separated by commas, in sequential order.
484          */
485          std::ostream& print(std::ostream& out) const {
486              if (is_empty()) {
487                  out << "<empty list>";
488              } else {
489                  out << "[";
490                  const_iterator start = begin();
491                  const_iterator fin = end();
492                  for (const_iterator iter = start; iter != fin; ++iter) {
493                      if (iter != start)
494                          out << ",";
495                      out << *iter;
496                  }
497                  out << "]";
498              }
499              return out;
500          }
501      protected:
502          bool validate_internal_integrity() {
503              //todo: fill this in
504              return true;
505          }
506      }; //end class SSLL
507  } // end namespace cop3530
508  #endif // _SSLL_H_
```

**PSLL checklist & source code**

# psll/checklist.txt

```
Pool-using Singly-Linked List written by Nickerson, Paul
COP 3530, 2014F 1087
=======================================================================
Part I:
=======================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

My LIST implementation 100% correctly supports the following methods
as described in part I:

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes


=======================================================================
Part II:
=======================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

======================================================================

```
Part III:
=======================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*
```

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My LIST implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Pool-using Singly-Linked List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt

# psll/source/PSLL.h

```
1   #ifndef _PSLL_H_
2   #define _PSLL_H_
3
4   // PSLL.H
5   //
6   // Pool-using Singly-linked list (non-polymorphic)
7   //
8   // Authors: Paul Nickerson, Dave Small
9   // for COP 3530
10  // 201409.16 - created
11
12  #include <iostream>
13  #include <stdexcept>
14  #include <cassert>
15  #include <string>
16
17  namespace cop3530 {
18      template <class T>
19      class PSLL {
20      private:
21          struct Node {
22              T item;
23              Node* next;
24              bool is_dummy;
25          }; // end struct Node
26          size_t num_main_list_items;
27          size_t num_free_list_items;
28          Node* head;
29          Node* tail;
30          Node* free_list_head;
31          Node* node_at(size_t position) const {
32              Node* n = head->next;
33              for (size_t i = 0; i != position; ++i, n = n->next);
34              return n;
35          }
36          Node* node_before(size_t position) const {
37              if (position == 0)
38                  return head;
39              else
40                  return node_at(position - 1);
41          }
42          Node* procure_free_node(bool force_allocation) {
43              Node* n;
44              if (force_allocation || free_list_size() == 0) {
45                  try {
46                      n = new Node();
47                  } catch (std::bad_alloc& ba) {
```

```cpp
48                        std::cerr << "procure_free_node(): failed to allocate new node"
                                 << std::endl;
49                        throw std::bad_alloc();
50                    }
51                } else {
52                    n = remove_node_after(free_list_head, num_free_list_items);
53                }
54                return n;
55            }
56            void shrink_pool_if_necessary() {
57                if (size() >= 100) {
58                    size_t old_size = size();
59                    while (free_list_size() > size() / 2) { //while the pool contains
                               more nodes than half the list size
60                        Node* n = remove_node_after(free_list_head, num_free_list_items);
61                        delete n;
62                    }
63                }
64            }

66            size_t free_list_size() { return num_free_list_items; }
67            Node* design_new_node(const T& element, Node* next = nullptr, bool dummy =
                     false, bool force_allocation = false) {
68                Node* n = procure_free_node(force_allocation);
69                n->is_dummy = dummy;
70                n->item = element;
71                n->next = next;
72                return n;
73            }
74            Node* design_new_node(Node* next = nullptr, bool dummy = false, bool
                      force_allocation = false) {
75                Node* n = procure_free_node(force_allocation);
76                n->is_dummy = dummy;
77                n->next = next;
78                return n;
79            }
80            void init() {
81                num_main_list_items = 0;
82                num_free_list_items = 0;
83                free_list_head = design_new_node(nullptr, true, true);
84                tail = design_new_node(nullptr, true, true);
85                head = design_new_node(tail, true, true);
86            }
87            void copy_constructor(const PSLL& src) {
88                //note: this function does *not* copy the free list
89                const_iterator fin = src.end();
90                for (const_iterator iter = src.begin(); iter != fin; ++iter) {
91                    push_back(*iter);
92                }
93                if ( ! src.size() == size())
94                    throw std::runtime_error("copy_constructor: Copying failed - sizes
                             don't match up");
```

```
 95          }
 96          Node* remove_node_after(Node* preceeding_node, size_t& list_size_counter) {
 97              if (preceeding_node->next == tail) {
 98                  throw std::runtime_error("remove_node_after:
                         preceeding_node->next==tail, and we cant remove the tail");
 99              }
100              if (preceeding_node == tail) {
101                  throw std::runtime_error("remove_node_after: preceeding_node==tail,
                         and we cant remove after the tail");
102              }
103              if (preceeding_node == free_list_head && free_list_size() == 0) {
104                  throw std::runtime_error("remove_node_after: attempt detected to
                         remove a node from an empty pool");
105              }
106              Node* removed_node = preceeding_node->next;
107              preceeding_node->next = removed_node->next;
108              removed_node->next = nullptr;
109              --list_size_counter;
110              return removed_node;
111          }
112
113          void insert_node_after(Node* existing_node, Node* new_node, size_t&
                 list_size_counter) {
114              new_node->next = existing_node->next;
115              existing_node->next = new_node;
116              ++list_size_counter;
117          }
118
119          //returns subsequent node's item and moves that node to the free pool
120          T remove_item_after(Node* preceeding_node) {
121              Node* removed_node = remove_node_after(preceeding_node,
                     num_main_list_items);
122              T item = removed_node->item;
123              insert_node_after(free_list_head, removed_node, num_free_list_items);
124              shrink_pool_if_necessary();
125              return item;
126          }
127
128      public:
129          //--------------------------------------------------
130          // iterators
131          //--------------------------------------------------
132          class PSLL_Iter: public std::iterator<std::forward_iterator_tag, T>
133          {
134          public:
135              // inheriting from std::iterator<std::forward_iterator_tag, T>
136              // automagically sets up these typedefs...
137              typedef T value_type;
138              typedef std::ptrdiff_t difference_type;
139              typedef T& reference;
140              typedef T* pointer;
141              typedef std::forward_iterator_tag iterator_category;
```

```
142
143             // but not these typedefs...
144             typedef PSLL_Iter self_type;
145             typedef PSLL_Iter& self_reference;
146
147         private:
148             Node* here;
149
150         public:
151             explicit PSLL_Iter(Node* start) : here(start) {
152                 if (start == nullptr)
153                     throw std::runtime_error("PSLL_Iter: start cannot be null");
154             }
155             PSLL_Iter(const PSLL_Iter& src) : here(src.here) {
156                 if (*this != src)
157                     throw std::runtime_error("PSLL_Iter: copy constructor failed");
158             }
159             reference operator*() const {
160                 return here->item;
161             }
162             pointer operator->() const {
163                 return & this->operator*();
164             }
165             self_reference operator=( const self_type& src ) {
166                 if (&src == this)
167                     return *this;
168                 here = src.here;
169                 if (*this != src)
170                     throw std::runtime_error("PSLL_Iter: copy assignment failed");
171                 return *this;
172             }
173             self_reference operator++() { // preincrement
174                 if (here->is_dummy)
175                     throw std::out_of_range("PSLL_Iter: Can't traverse past the end
                         of the list");
176                 here = here->next;
177                 return *this;
178             }
179             self_type operator++(int) { // postincrement
180                 self_type t(*this); //save state
181                 operator++(); //apply increment
182                 return t; //return state held before increment
183             }
184             bool operator==(const self_type& rhs) const {
185                 return rhs.here == here;
186             }
187             bool operator!=(const self_type& rhs) const {
188                 return ! operator==(rhs);
189             }
190         };
191
192         class PSLL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
```

```
193            {
194        public:
195                // inheriting from std::iterator<std::forward_iterator_tag, T>
196                // automagically sets up these typedefs...
197                typedef T value_type;
198                typedef std::ptrdiff_t difference_type;
199                typedef const T& reference;
200                typedef const T* pointer;
201                typedef std::forward_iterator_tag iterator_category;
202
203                // but not these typedefs...
204                typedef PSLL_Const_Iter self_type;
205                typedef PSLL_Const_Iter& self_reference;
206
207        private:
208                const Node* here;
209
210        public:
211                explicit PSLL_Const_Iter(Node* start) : here(start) {
212                    if (start == nullptr)
213                        throw std::runtime_error("PSLL_Const_Iter: start cannot be null");
214                }
215                PSLL_Const_Iter(const PSLL_Const_Iter& src) : here(src.here) {
216                    if (*this != src)
217                        throw std::runtime_error("PSLL_Const_Iter: copy constructor
                             failed");
218                }
219
220                reference operator*() const {
221                    return here->item;
222                }
223                pointer operator->() const {
224                    return & this->operator*();
225                }
226                self_reference operator=( const self_type& src ) {
227                    if (&src == this)
228                        return *this;
229                    here = src.here;
230                    if (*this != src)
231                        throw std::runtime_error("PSLL_Const_Iter: copy assignment
                             failed");
232                    return *this;
233                }
234                self_reference operator++() { // preincrement
235                    if (here->is_dummy)
236                        throw std::out_of_range("PSLL_Const_Iter: Can't traverse past the
                             end of the list");
237                    here = here->next;
238                    return *this;
239                }
240                self_type operator++(int) { // postincrement
241                    self_type t(*this); //save state
```

```cpp
                operator++(); //apply increment
                return t; //return state held before increment
            }
            bool operator==(const self_type& rhs) const {
                return rhs.here == here;
            }
            bool operator!=(const self_type& rhs) const {
                return ! operator==(rhs);
            }
        };


        //--------------------------------------------------
        // types
        //--------------------------------------------------
        /*typedef std::size_t size_t;*/
        typedef T value_type;
        typedef PSLL_Iter iterator;
        typedef PSLL_Const_Iter const_iterator;

        iterator begin() {
            return iterator(head->next);
        }
        iterator end() {
            return iterator(tail);
        }
        /*
            Note to self: the following overloads will fail if not defined as const
        */
        const_iterator begin() const {
            return const_iterator(head->next);
        }
        const_iterator end() const {
            return const_iterator(tail);
        }


        //--------------------------------------------------
        // operators
        //--------------------------------------------------
        T& operator[](size_t i) {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
            }
            return node_at(i)->item;
        }

        const T& operator[](size_t i) const {
            if (i >= size()) {
                throw std::out_of_range(std::string("operator[]: No element at
                    position ") + std::to_string(i));
            }
            return node_at(i)->item;
```

```
292            }
293
294            //--------------------------------------------------
295            // Constructors/destructor/assignment operator
296            //--------------------------------------------------
297
298            PSLL() {
299                init();
300            }
301            //--------------------------------------------------
302            //copy constructor
303            PSLL(const PSLL& src) {
304                init();
305                copy_constructor(src);
306            }
307
308            //--------------------------------------------------
309            //destructor
310            ~PSLL() {
311                // safely dispose of this PSLL's contents
312                clear();
313            }
314
315            //--------------------------------------------------
316            //copy assignment constructor
317            PSLL& operator=(const PSLL& src) {
318                if (&src == this) // check for self-assignment
319                    return *this;   // do nothing
320                // safely dispose of this PSLL's contents
321                clear();
322                // populate this PSLL with copies of the other PSLL's contents
323                copy_constructor(src);
324                return *this;
325            }
326
327            //--------------------------------------------------
328            // member functions
329            //--------------------------------------------------
330
331            /*
332                replaces the existing element at the specified position with the
                        specified element and
333                returns the original element.
334            */
335            T replace(const T& element, size_t position) {
336                T old_item;
337                if (position >= size()) {
338                    throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
339                } else {
340                    //we are guaranteed to be at a non-dummy item now because of the
                            above if statement
```

```cpp
                Node* iter = node_at(position);
                old_item = iter->item;
                iter->item = element;
            }
            return old_item;
        }

        //--------------------------------------------------
        /*
            adds the specified element to the list at the specified position,
                shifting the element
            originally at that and those in subsequent positions one position to the
                right.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of
                    the list: ") + std::to_string(position));
            } else if (position == size()) {
                //special O(1) case
                push_back(element);
            } else {
                //node_before_position is guaranteed to point to a valid node
                    because we use a dummy head node
                Node* node_before_position = node_before(position);
                Node* node_at_position = node_before_position->next;
                Node* new_node;
                try {
                    new_node = design_new_node(element, node_at_position);
                } catch (std::bad_alloc& ba) {
                    std::cerr << "insert(): failed to allocate memory for new node"
                        << std::endl;
                    throw std::bad_alloc();
                }
                insert_node_after(node_before_position, new_node,
                    num_main_list_items);
            }
        }

        //--------------------------------------------------
        //Note to self: use reference here because we receive the original object
            instance,
        //then copy it into n->item so we have it if the original element goes out
            of scope
        /*
            prepends the specified element to the list.
        */
        void push_front(const T& element) {
            insert(element, 0);
        }

        //--------------------------------------------------
```

```
385        /*
386            appends the specified element to the list.
387        */
388        void push_back(const T& element) {
389            Node* new_tail;
390            try {
391                new_tail = design_new_node(nullptr, true);
392            } catch (std::bad_alloc& ba) {
393                std::cerr << "push_back(): failed to allocate memory for new tail"
394                    << std::endl;
394                throw std::bad_alloc();
395            }
396            insert_node_after(tail, new_tail, num_main_list_items);
397            //transform the current tail node from a dummy to a real node holding
                     element
398            tail->is_dummy = false;
399            tail->item = element;
400            tail->next = new_tail;
401            tail = tail->next;
402        }

404        //---------------------------------------------------
405        //Note to self: no reference here, so we get our copy of the item, then
                 return a copy
406        //of that so the client still has a valid instance if our destructor is
                 called
407        /*
408            removes and returns the element at the list's head.
409        */
410        T pop_front() {
411            if (is_empty()) {
412                throw std::out_of_range("pop_front: Can't pop: list is empty");
413            }
414            if (head->next == tail) {
415                throw std::runtime_error("pop_front: head->next == tail, but list
                     says it's not empty (corrupt state)");
416            }
417            return remove_item_after(head);
418        }

420        //---------------------------------------------------
421        /*
422            removes and returns the element at the list's tail.
423        */
424        T pop_back() {
425            if (is_empty()) {
426                throw std::out_of_range("pop_back: Can't pop: list is empty");
427            }
428            if (head->next == tail) {
429                throw std::runtime_error("pop_back: head->next == tail, but list
                     says it's not empty (corrupt state)");
430            }
```

```
            //XXX this is O(N), a disadvantage of this architecture
            Node* node_before_last = node_before(size() - 1);
            T item = remove_item_after(node_before_last);
            return item;
        }

        //---------------------------------------------------
        /*
            removes and returns the the element at the specified position,
            shifting the subsequent elements one position to the left.
        */
        T remove(size_t position) {
            T item;
            if (position >= size()) {
                throw std::out_of_range(std::string("remove: No element at position
                    ") + std::to_string(position));
            }
            if (head->next == tail) {
                throw std::runtime_error("remove: head->next == tail, but list says
                    it's not empty (corrupt state)");
            }
            //using a dummy head node guarantees that there be a node immediately
                    preceeding the specified position
            Node *node_before_position = node_before(position);
            item = remove_item_after(node_before_position);
            return item;
        }
        //---------------------------------------------------
        /*
            returns (without removing from the list) the element at the specified
                position.
        */
        T item_at(size_t position) const {
            if (position >= size()) {
                throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
            }
            return operator[](position);
        }


        //---------------------------------------------------
        /*
            returns true IFF the list contains no elements.
        */
        bool is_empty() const {
            return size() == 0;
        }

        //---------------------------------------------------
        /*
            returns the number of elements in the list.
```

```
478         */
479         size_t size() const {
480             if (num_main_list_items == 0 && head->next != tail) {
481                 throw std::runtime_error("size: head->next != tail, but list says
                        it's empty (corrupt state)");
482             } else if (num_main_list_items > 0 && head->next == tail) {
483                 throw std::runtime_error("size: head->next == tail, but list says
                        it's not empty (corrupt state)");
484             }
485             return num_main_list_items;
486         }
487
488         //-------------------------------------------------
489         /*
490             removes all elements from the list.
491         */
492         void clear() {
493             while (size()) {
494                 pop_front();
495             }
496         }
497         //-------------------------------------------------
498         /*
499             returns true IFF one of the elements of the list matches the specified
                element.
500         */
501         bool contains(const T& element,
502                 bool equals(const T& a, const T& b)) const {
503             bool element_in_list = false;
504             const_iterator fin = end();
505             for (const_iterator iter = begin(); iter != fin; ++iter) {
506                 if (equals(*iter, element)) {
507                     element_in_list = true;
508                     break;
509                 }
510             }
511             return element_in_list;
512         }
513
514         //-------------------------------------------------
515         /*
516             If the list is empty, inserts "<empty list>" into the ostream;
517             otherwise, inserts, enclosed in square brackets, the list's elements,
518             separated by commas, in sequential order.
519         */
520         std::ostream& print(std::ostream& out) const {
521             if (is_empty()) {
522                 out << "<empty list>";
523             } else {
524                 out << "[";
525                 const_iterator start = begin();
526                 const_iterator fin = end();
```

```cpp
            for (const_iterator iter = start; iter != fin; ++iter) {
                if (iter != start)
                    out << ",";
                out << *iter;
            }
            out << "]";
        }
        return out;
    }
protected:
    bool validate_internal_integrity() {
        //todo: fill this in
        return true;
    }
}; //end class PSLL
} // end namespace cop3530
#endif // _PSLL_H_
```

# SDAL checklist & source code

# sdal/checklist.txt

```
Simple Dynamic Array-based List written by Nickerson, Paul
COP 3530, 2014F 1087
=======================================================================
Part I:
=======================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

My LIST implementation 100% correctly supports the following methods
as described in part I:

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes


=======================================================================
Part II:
=======================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes


=======================================================================

Part III:
=======================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
    typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My LIST implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Simple Dynamic Array-based List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt

# sdal/source/SDAL.h

SDAL.h

```
1   #ifndef _SDAL_H_
2   #define _SDAL_H_
3
4   // SDAL.H
5   //
6   // Singly-linked list (non-polymorphic)
7   //
8   // Authors: Paul Nickerson, Dave Small
9   // for COP 3530
10  // 201409.16 - created
11
12  #include <iostream>
13  #include <stdexcept>
14  #include <cassert>
15  #include <memory>
16  #include <string>
17  #include <cmath>
18
19  namespace cop3530 {
20      template <class T>
21      class SDAL {
22      private:
23          T* item_array;
24          //XXX: do these both need to be size_t?
25          size_t array_size;
26          size_t num_items;
27          size_t embiggen_counter = 0;
28          size_t shrink_counter = 0;
29          T* allocate_nodes(size_t quantity) {
30              try {
31                  T* new_item_array = new T[quantity];
32                  return new_item_array;
33              } catch (std::bad_alloc& ba) {
34                  std::cerr << "allocate_nodes(): failed to allocate item array of
                        size " << quantity << std::endl;
35                  throw std::bad_alloc();
36              }
37          }
38          void embiggen_if_necessary() {
39              /*
40                  Whenever an item is added and the backing array is full, allocate a
                        new array 150% the size
41                  of the original, copy the items over to the new array, and
                        deallocate the original one.
42              */
43              size_t filled_slots = size();
44              if (filled_slots == array_size) {
```

83

```cpp
                size_t new_array_size = ceil(array_size * 1.5);
                T* new_item_array = allocate_nodes(new_array_size);
                for (size_t i = 0; i != filled_slots; ++i) {
                    new_item_array[i] = item_array[i];
                }
                delete[] item_array;
                item_array = new_item_array;
                array_size = new_array_size;
                ++embiggen_counter;
            }
        }
        void shrink_if_necessary() {
            /*
                Because we don't want the list to waste too much memory, whenever
                    the array's size is  100 slots
                and fewer than half the slots are used, allocate a new array 50% the
                    size of the original, copy
                the items over to the new array, and deallocate the original one.
            */
            size_t filled_slots = size();
            if (array_size >= 100 && filled_slots < array_size / 2) {
                size_t new_array_size = ceil(array_size * 0.5);
                T* new_item_array = allocate_nodes(new_array_size);
                for (size_t i = 0; i != filled_slots; ++i) {
                    new_item_array[i] = item_array[i];
                }
                delete[] item_array;
                item_array = new_item_array;
                array_size = new_array_size;
                ++shrink_counter;
            }
        }
        void init(size_t num_nodes_to_preallocate) {
            array_size = num_nodes_to_preallocate;
            num_items = 0;
            item_array = allocate_nodes(array_size);
        }
        void copy_constructor(const SDAL& src) {
            const_iterator fin = src.end();
            for (const_iterator iter = src.begin(); iter != fin; ++iter) {
                push_back(*iter);
            }
            if ( ! src.size() == size())
                throw std::runtime_error("copy_constructor: Copying failed - sizes
                    don't match up");
        }
    public:

        //----------------------------------------------------
        // iterators
        //----------------------------------------------------
        class SDAL_Iter: public std::iterator<std::forward_iterator_tag, T>
```

```
94              {
95          public:
96                  // inheriting from std::iterator<std::forward_iterator_tag, T>
97                  // automagically sets up these typedefs...
98                  //todo: figure out why we cant comment these out, which we should be
                        able to if they were
99                  //defined when inheriting
100                 typedef T value_type;
101                 typedef std::ptrdiff_t difference_type;
102                 typedef T& reference;
103                 typedef T* pointer;
104                 typedef std::forward_iterator_tag iterator_category;

106                 // but not these typedefs...
107                 typedef SDAL_Iter self_type;
108                 typedef SDAL_Iter& self_reference;

110         private:
111                 T* iter;
112                 T* end_iter;

114         public:
115                 explicit SDAL_Iter(T* item_array, T* end_ptr): iter(item_array),
                        end_iter(end_ptr) {
116                     if (item_array == nullptr)
117                         throw std::runtime_error("SDAL_Iter: item_array cannot be null");
118                     if (end_ptr == nullptr)
119                         throw std::runtime_error("SDAL_Iter: end_ptr cannot be null");
120                     if (item_array > end_ptr)
121                         throw std::runtime_error("SDAL_Iter: item_array pointer cannot be
                            past end_ptr");
122                 }
123                 SDAL_Iter(const SDAL_Iter& src): iter(src.iter), end_iter(src.end_iter) {
124                     if (*this != src)
125                         throw std::runtime_error("SDAL_Iter: copy constructor failed");
126                 }
127                 reference operator*() const {
128                     return *iter;
129                 }
130                 pointer operator->() const {
131                     return & this->operator*();
132                 }
133                 self_reference operator=( const self_type& src ) {
134                     if (&src == this)
135                         return *this;
136                     iter = src.iter;
137                     end_iter = src.end_iter;
138                     if (*this != src)
139                         throw std::runtime_error("SDAL_Iter: copy assignment failed");
140                     return *this;
141                 }
142                 self_reference operator++() { // preincrement
```

```
143             if (iter == end_iter)
144                 throw std::out_of_range("SDAL_Iter: Can't traverse past the end
                        of the list");
145             ++iter;
146             return *this;
147         }
148         self_type operator++(int) { // postincrement
149             self_type t(*this); //save state
150             operator++(); //apply increment
151             return t; //return state held before increment
152         }
153         bool operator==(const self_type& rhs) const {
154             return rhs.iter == iter && rhs.end_iter == end_iter;
155         }
156         bool operator!=(const self_type& rhs) const {
157             return ! operator==(rhs);
158         }
159     };
160
161     class SDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T>
162     {
163     public:
164         // inheriting from std::iterator<std::forward_iterator_tag, T>
165         // automagically sets up these typedefs...
166         typedef T value_type;
167         typedef std::ptrdiff_t difference_type;
168         typedef const T& reference;
169         typedef const T* pointer;
170         typedef std::forward_iterator_tag iterator_category;
171
172         // but not these typedefs...
173         typedef SDAL_Const_Iter self_type;
174         typedef SDAL_Const_Iter& self_reference;
175     private:
176         T* iter;
177         T* end_iter;
178     public:
179         explicit SDAL_Const_Iter(T* item_array, T* end_ptr): iter(item_array),
                end_iter(end_ptr) {
180             if (item_array == nullptr)
181                 throw std::runtime_error("SDAL_Const_Iter: item_array cannot be
                        null");
182             if (end_ptr == nullptr)
183                 throw std::runtime_error("SDAL_Const_Iter: end_ptr cannot be
                        null");
184             if (item_array > end_ptr)
185                 throw std::runtime_error("SDAL_Const_Iter: item_array pointer
                        cannot be past end_ptr");
186         }
187         SDAL_Const_Iter(const SDAL_Const_Iter& src): iter(src.iter),
                end_iter(src.end_iter) {
188             if (*this != src)
```

```
189                     throw std::runtime_error("SDAL_Const_Iter: copy constructor
                            failed");
190             }
191         reference operator*() const {
192             return *iter;
193         }
194         pointer operator->() const {
195             return & this->operator*();
196         }
197         self_reference operator=( const self_type& src ) {
198             if (&src == this)
199                 return *this;
200             iter = src.iter;
201             end_iter = src.end_iter;
202             if (*this != src)
203                 throw std::runtime_error("SDAL_Const_Iter: copy assignment
                            failed");
204             return *this;
205         }
206         self_reference operator++() { // preincrement
207             if (iter == end_iter)
208                 throw std::out_of_range("SDAL_Const_Iter: Can't traverse past the
                            end of the list");
209             ++iter;
210             return *this;
211         }
212         self_type operator++(int) { // postincrement
213             self_type t(*this); //save state
214             operator++(); //apply increment
215             return t; //return state held before increment
216         }
217         bool operator==(const self_type& rhs) const {
218             return rhs.iter == iter && rhs.end_iter == end_iter;
219         }
220         bool operator!=(const self_type& rhs) const {
221             return ! operator==(rhs);
222         }
223     };

224

225     //--------------------------------------------------
226     // types
227     //--------------------------------------------------
228     typedef T value_type;
229     typedef SDAL_Iter iterator;
230     typedef SDAL_Const_Iter const_iterator;

231

232     iterator begin() { return SDAL_Iter(item_array, item_array + num_items); }
233     iterator end() { return SDAL_Iter(item_array + num_items, item_array +
            num_items); }

234

235     const_iterator begin() const { return SDAL_Const_Iter(item_array,
            item_array + num_items); }
```

```
236         const_iterator end() const { return SDAL_Const_Iter(item_array + num_items,
                item_array + num_items); }
237
238         //----------------------------------------------------
239         // operators
240         //----------------------------------------------------
241         T& operator[](size_t i) {
242             if (i >= size()) {
243                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
244             }
245             return item_array[i];
246         }
247
248         const T& operator[](size_t i) const {
249             if (i >= size()) {
250                 throw std::out_of_range(std::string("operator[]: No element at
                        position ") + std::to_string(i));
251             }
252             return item_array[i];
253         }
254
255         //----------------------------------------------------
256         // Constructors/destructor/assignment operator
257         //----------------------------------------------------
258
259         SDAL(size_t num_nodes_to_preallocate = 50) {
260             init(num_nodes_to_preallocate);
261         }
262
263         //----------------------------------------------------
264         //copy constructor
265         SDAL(const SDAL& src): SDAL(src.array_size) {
266             init(src.array_size);
267             copy_constructor(src);
268         }
269
270         //----------------------------------------------------
271         //destructor
272         ~SDAL() {
273             // safely dispose of this SDAL's contents
274             delete[] item_array;
275         }
276
277         //----------------------------------------------------
278         //copy assignment constructor
279         SDAL& operator=(const SDAL& src) {
280             if (&src == this) // check for self-assignment
281                 return *this;   // do nothing
282             delete[] item_array;
283             init(src.array_size);
284             copy_constructor(src);
```

```cpp
                return *this;
            }

            //----------------------------------------------------
            // member functions
            //----------------------------------------------------

            /*
                replaces the existing element at the specified position with the
                    specified element and
                returns the original element.
            */
            T replace(const T& element, size_t position) {
                T old_item;
                if (position >= size()) {
                    throw std::out_of_range(std::string("replace: No element at position
                        ") + std::to_string(position));
                } else {
                    old_item = item_array[position];
                    item_array[position] = element;
                }
                return old_item;
            }

            //----------------------------------------------------
            /*
                adds the specified element to the list at the specified position,
                    shifting the element
                originally at that and those in subsequent positions one position to the
                    right.
            */
            void insert(const T& element, size_t position) {
                if (position > size()) {
                    throw std::out_of_range(std::string("insert: Position is outside of
                        the list: ") + std::to_string(position));
                } else {
                    embiggen_if_necessary();
                    //shift remaining items right
                    for (size_t i = size(); i != position; --i) {
                        item_array[i] = item_array[i - 1];
                    }
                    item_array[position] = element;
                    ++num_items;
                }
            }

            //----------------------------------------------------
            //Note to self: use reference here because we receive the original object
                instance,
            //then copy it into n->item so we have it if the original element goes out
                of scope
            /*
```

```
                    prepends the specified element to the list.
             */
            void push_front(const T& element) {
                insert(element, 0);
            }

            //---------------------------------------------------
            /*
                appends the specified element to the list.
             */
            void push_back(const T& element) {
                insert(element, size());
            }


            //---------------------------------------------------
            //Note to self: no reference here, so we get our copy of the item, then
                  return a copy
            //of that so the client still has a valid instance if our destructor is
                  called
            /*
                removes and returns the element at the list's head.
             */
            T pop_front() {
                if (is_empty()) {
                    throw std::out_of_range("pop_front: Can't pop: list is empty");
                }
                return remove(0);
            }

            //---------------------------------------------------
            /*
                removes and returns the element at the list's tail.
             */
            T pop_back() {
                if (is_empty()) {
                    throw std::out_of_range("pop_back: Can't pop: list is empty");
                }
                return remove(size() - 1);
            }

            //---------------------------------------------------
            /*
                removes and returns the the element at the specified position,
                shifting the subsequent elements one position to the left.
             */
            T remove(size_t position) {
                T item;
                if (position >= size()) {
                    throw std::out_of_range(std::string("remove: No element at position
                          ") + std::to_string(position));
                } else {
```

```
            item = item_array[position];
            //shift remaining items left
            for (size_t i = position + 1; i != size(); ++i) {
                item_array[i - 1] = item_array[i];
            }
            --num_items;
            shrink_if_necessary();
        }
        return item;
    }

    //--------------------------------------------------
    /*
        returns (without removing from the list) the element at the specified
            position.
    */
    T item_at(size_t position) const {
        if (position >= size()) {
            throw std::out_of_range(std::string("item_at: No element at position
                ") + std::to_string(position));
        }
        return operator[](position);
    }

    //--------------------------------------------------
    /*
        returns true IFF the list contains no elements.
    */
    bool is_empty() const {
        return size() == 0;
    }

    //--------------------------------------------------
    /*
        returns the number of elements in the list.
    */
    size_t size() const {
        return num_items;
    }

    //--------------------------------------------------
    /*
        removes all elements from the list.
    */
    void clear() {
        //no reason to do memory deallocation here, just overwrite the old items
            later and save
        //deallocation for the deconstructor
        num_items = 0;
    }

    //--------------------------------------------------
```

```
428          /*
429              returns true IFF one of the elements of the list matches the specified
                     element.
430          */
431          bool contains(const T& element,
432                  bool equals(const T& a, const T& b)) const {
433              bool element_in_list = false;
434              const_iterator fin = end();
435              for (const_iterator iter = begin(); iter != fin; ++iter) {
436                  if (equals(*iter, element)) {
437                      element_in_list = true;
438                      break;
439                  }
440              }
441              return element_in_list;
442          }

444          //--------------------------------------------------
445          /*
446              If the list is empty, inserts "<empty list>" into the ostream;
447              otherwise, inserts, enclosed in square brackets, the list's elements,
448              separated by commas, in sequential order.
449          */
450          std::ostream& print(std::ostream& out) const {
451              if (is_empty()) {
452                  out << "<empty list>";
453              } else {
454                  out << "[";
455                  const_iterator start = begin();
456                  const_iterator fin = end();
457                  for (const_iterator iter = start; iter != fin; ++iter) {
458                      if (iter != start)
459                          out << ",";
460                      out << *iter;
461                  }
462                  out << "]";
463              }
464              return out;
465          }
466      protected:
467          bool validate_internal_integrity() {
468              //todo: fill this in
469              return true;
470          }
471      };
472  } // end namespace cop3530

474  #endif // _SDAL_H_
```

**CDAL checklist & source code**

# cdal/checklist.txt

```
Chained Dynamic Array-based List written by Nickerson, Paul
COP 3530, 2014F 1087
======================================================================
Part I:
======================================================================
My LIST implementation uses the data structure described in the part I
instructions and conforms to the technique required for this list
variety: yes

My LIST implementation 100% correctly supports the following methods
as described in part I:

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes


======================================================================
Part II:
======================================================================
My LIST implementation 100% correctly supports the following methods
as described in part II:

* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes

My LIST implementation 100% correctly supports the following data
members as described in part II:

* size_t
* value_type
* iterator
* const_iterator

My ITERATOR implementation 100% correctly supports the following
```

methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes

My CONST ITERATOR implementation 100% correctly supports the following
methods as described in part II:

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My CONST ITERATOR implementation 100% correctly supports the following
data members as described in part II:

* value_type: yes
* difference_type: yes
* reference: yes
* pointer: yes
* iterator_category: yes
* self_type: yes
* self_reference: yes


========================================================================

Part III:
=======================================================================
My LIST implementation 100% correctly supports the following
methods as described in part III:

* operator[]: yes
* operator[] const: yes

For my LIST's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* replace: yes
* insert: yes
* push_back: yes
* push_front: yes
* remove: yes
* pop_back: yes
* pop_front: yes
* item_at: yes
* is_empty: yes
* clear: yes
* contains: yes
* print: yes
* size: yes
* begin (returning an iterator): yes
* end (returning an iterator): yes
* begin (returning a const iterator): yes
* end (returning an const iterator): yes
* operator[]: yes
* operator[] const: yes

For my ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

For my CONST ITERATOR's methods

- I wrote documentation identifying the complete behavior (both normal
  and exceptional) of the method, *AND*

- when something unexpected occurs, the method throws appropriately
  typed exceptions, *AND*

- my implementation behaves 100% precisely as documented, *AND*

- I have proven this by creating a suite of CATCH unit tests for
  the method to verify that the method behaves as documented, *AND*

- the method passes all of those unit tests.

* constructor: yes
* explicit constructor: yes
* operator*: yes
* operator-: no
* operator=: yes
* operator++ (pre): yes
* operator++ (post): yes
* operator==: yes
* operator!=: yes

My LIST implementation compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS compiles correctly using g++ v4.8.2 on the
OpenBSD VM: yes

My UNIT TESTS run correctly on the OpenBSD VM: yes

I affirm that all the responsess I have provided above are 100% true.
Should it be determined that any are not 100% true, I agree to take a 0
(zero) on the assignment: yes

I affirm that I am the sole author of this Chained Dynamic Array-based List
and the associated unit tests.
Paul Nickerson, 11/24/2014 in COP3530 section 1087

=====================================================================
In addition to the unit tests, the old_tests directory contains a fuzzer
which stress-tests every list and compares their states to ensure they
all behave equivalently as well as maintain internal integrity.
=====================================================================

How to compile and run my unit tests on the OpenBSD VM
cd list_source_directory
./compile.sh
./unit_tester -s > output.txt

# cdal/source/CDAL.h

```
1   #ifndef _CDAL_H_
2   #define _CDAL_H_
3
4   // CDAL.H
5   //
6   // Chained Dynamic Array-based List (non-polymorphic)
7   //
8   // Authors: Paul Nickerson, Dave Small
9   // for COP 3530
10  // 201409.16 - created
11
12  #include <iostream>
13  #include <stdexcept>
14  #include <cassert>
15  #include <math.h>
16
17  namespace cop3530 {
18      template <class T>
19      class CDAL {
20      private:
21          struct Node {
22              //Node is an element in the linked list and contains an array of items
23              T* item_array;
24              Node* next;
25              bool is_dummy;
26          };
27          struct ItemLoc {
28              //ItemLoc describes the position of an item, including its linked list
29                      node and position within the array held by that node
30              Node* node;
31              size_t array_index;
32              T& item_ref;
33          };
34          size_t num_items;
35          size_t num_available_nodes; //excludes head/tail nodes
36          size_t embiggen_counter = 0;
37          size_t shrink_counter = 0;
38          Node* head;
39          Node* tail;
40          static const size_t array_size = 50; //length of each chained array
41          Node* node_at(size_t position) const {
42              Node* n = head->next;
43              for (size_t i = 0; i != position; ++i, n = n->next);
44              return n;
45          }
46          Node* node_before(size_t position) const {
47              if (position == 0)
```

```cpp
47              return head;
48          else
49              return node_at(position - 1);
50      }
51
52      ItemLoc loc_from_pos(size_t position) const {
53          size_t node_position = floor(position / array_size);
54          Node* n = node_at(node_position);
55          size_t array_index = position % array_size;
56          ItemLoc loc {n, array_index, n->item_array[array_index]};
57          return loc;
58      }
59
60      Node* design_new_node(Node* next = nullptr, bool dummy = false) const {
61          Node* n;
62          try {
63              n = new Node();
64          } catch (std::bad_alloc& ba) {
65              std::cerr << "design_new_node(): failed to allocate memory for new
                      node" << std::endl;
66              throw std::bad_alloc();
67          }
68          n->is_dummy = dummy;
69          try {
70              n->item_array = new T[array_size];
71          } catch (std::bad_alloc& ba) {
72              std::cerr << "design_new_node(): failed to allocate memory for item
                      array" << std::endl;
73              throw std::bad_alloc();
74          }
75          n->next = next;
76          return n;
77      }
78
79      void init() {
80          num_items = 0;
81          num_available_nodes = 0;
82          tail = design_new_node(nullptr, true);
83          head = design_new_node(tail, true);
84      }
85
86      void free_node(Node* n) {
87          delete[] n->item_array;
88          delete n;
89      }
90
91      void drop_node_after(Node* n) {
92          assert(n->next != tail);
93          Node* removed_node = n->next;
94          n->next = removed_node->next;
95          free_node(removed_node);
96          --num_available_nodes;
```

```
 97            }
 98
 99            size_t num_used_nodes() {
100                return ceil(size() / array_size);
101            }
102
103            void embiggen_if_necessary() {
104                //embiggen is a perfectly cromulent word
105                /*
106                    If each array slot in every link is filled and we want to add a new
107                        item, allocate and append a new link
108                */
109                if (size() == num_available_nodes * array_size) {
110                    //transform tail into a regular node and append a new tail
111                    Node* n = tail;
112                    n->is_dummy = false;
113                    tail = n->next = design_new_node(nullptr, true);
114                    ++num_available_nodes;
115                    ++embiggen_counter;
116                }
117            }
118
119            void shrink_if_necessary() {
120                /*
121                    Because we don't want the list to waste too much memory, whenever
122                        the more than half of the arrays
123                    are unused (they would all be at the end of the chain), deallocate
124                        half the unused arrays.
125                */
126                size_t used = num_used_nodes();
127                size_t num_unused_nodes = num_available_nodes - used;
128                if (num_unused_nodes > used) {
129                    size_t nodes_to_keep = used + ceil(num_unused_nodes * 0.5);
130                    Node* last_node = node_before(nodes_to_keep);
131                    while (last_node->next != tail) {
132                        drop_node_after(last_node);
133                    }
134                    ++shrink_counter;
135                }
136            }
137            void copy_constructor(const CDAL& src) {
138                const_iterator fin = src.end();
139                for (const_iterator iter = src.begin(); iter != fin; ++iter) {
140                    push_back(*iter);
141                }
142                if ( ! src.size() == size())
143                    throw std::runtime_error("copy_constructor: Copying failed - sizes
144                        don't match up");
145            }
146
147    public:
148        //------------------------------------------------
```

```
 97            }
 98
 99            size_t num_used_nodes() {
100                return ceil(size() / array_size);
101            }
102
103            void embiggen_if_necessary() {
104                //embiggen is a perfectly cromulent word
105                /*
106                    If each array slot in every link is filled and we want to add a new
                        item, allocate and append a new link
107                */
108                if (size() == num_available_nodes * array_size) {
109                    //transform tail into a regular node and append a new tail
110                    Node* n = tail;
111                    n->is_dummy = false;
112                    tail = n->next = design_new_node(nullptr, true);
113                    ++num_available_nodes;
114                    ++embiggen_counter;
115                }
116            }
117
118            void shrink_if_necessary() {
119                /*
120                    Because we don't want the list to waste too much memory, whenever
                        the more than half of the arrays
121                    are unused (they would all be at the end of the chain), deallocate
                        half the unused arrays.
122                */
123                size_t used = num_used_nodes();
124                size_t num_unused_nodes = num_available_nodes - used;
125                if (num_unused_nodes > used) {
126                    size_t nodes_to_keep = used + ceil(num_unused_nodes * 0.5);
127                    Node* last_node = node_before(nodes_to_keep);
128                    while (last_node->next != tail) {
129                        drop_node_after(last_node);
130                    }
131                    ++shrink_counter;
132                }
133            }
134        void copy_constructor(const CDAL& src) {
135            const_iterator fin = src.end();
136            for (const_iterator iter = src.begin(); iter != fin; ++iter) {
137                push_back(*iter);
138            }
139            if ( ! src.size() == size())
140                throw std::runtime_error("copy_constructor: Copying failed - sizes
                    don't match up");
141        }
142
143    public:
144        //------------------------------------------------
```

```
145        // iterators
146        //--------------------------------------------------
147        class CDAL_Iter: public std::iterator<std::forward_iterator_tag, T> {
148        private:
149            Node* curr_node;
150            size_t curr_array_index;
151        public:
152            typedef std::ptrdiff_t difference_type;
153            typedef T& reference;
154            typedef T* pointer;
155            typedef std::forward_iterator_tag iterator_category;
156            typedef T value_type;
157            typedef CDAL_Iter self_type;
158            typedef CDAL_Iter& self_reference;
159
160            //need copy constructor/assigner to make this a first class ADT (doesn't
161                hold pointers that need freeing)
            CDAL_Iter(ItemLoc const& here):
162                curr_node(here.node),
163                curr_array_index(here.array_index)
164            {}
165            CDAL_Iter(const self_type& src):
166                curr_node(src.curr_node),
167                curr_array_index(src.curr_array_index)
168            {
169                if (*this != src)
170                    throw std::runtime_error("CDAL_Iter: copy constructor failed");
171            }
172            self_reference operator=(const self_type& rhs) {
173                //copy assigner
174                if (&rhs == this) return *this;
175                curr_node = rhs.curr_node;
176                curr_array_index = rhs.curr_array_index;
177                if (*this != src)
178                    throw std::runtime_error("CDAL_Iter: copy assignment failed");
179                return this;
180            }
181            self_reference operator++() { // preincrement
182                if (curr_node->is_dummy)
183                    throw std::out_of_range("CDAL_Iter: Can't traverse past the end
                        of the list");
184                curr_array_index = (curr_array_index + 1) % array_size;
185                if (curr_array_index == 0) curr_node = curr_node->next;
186                return *this;
187            }
188            self_type operator++(int) { // postincrement
189                self_type t(*this); //save state
190                operator++(); //apply increment
191                return t; //return state held before increment
192            }
193            reference operator*() const {
194                return curr_node->item_array[curr_array_index];
```

```
195                }
196                pointer operator->() const {
197                    return & this->operator*();
198                }
199                bool operator==(const self_type& rhs) const {
200                    return rhs.curr_node == curr_node
201                            && rhs.curr_array_index == curr_array_index;
202                }
203                bool operator!=(const self_type& rhs) const {
204                    return ! operator==(rhs);
205                }
206            };

207
208            class CDAL_Const_Iter: public std::iterator<std::forward_iterator_tag, T> {
209            private:
210                Node* curr_node;
211                size_t curr_array_index;
212            public:
213                typedef const T value_type;
214                typedef const T& reference;
215                typedef const T* pointer;
216                typedef std::forward_iterator_tag iterator_category;
217                typedef std::ptrdiff_t difference_type;
218                typedef CDAL_Const_Iter self_type;
219                typedef CDAL_Const_Iter& self_reference;

220
221                //need copy constructor/assigner to make this a first class ADT (doesn't
                        hold pointers that need freeing)
222                CDAL_Const_Iter(ItemLoc const& here):
223                    curr_node(here.node),
224                    curr_array_index(here.array_index)
225                {}
226                CDAL_Const_Iter(const self_type& src):
227                    curr_node(src.curr_node),
228                    curr_array_index(src.curr_array_index)
229                {
230                    if (*this != src)
231                        throw std::runtime_error("CDAL_Iter: copy constructor failed");
232                }
233                self_reference operator=(const self_type& rhs) {
234                    //copy assigner
235                    if (&rhs == this) return *this;
236                    curr_node = rhs.curr_node;
237                    curr_array_index = rhs.curr_array_index;
238                    if (*this != src)
239                        throw std::runtime_error("CDAL_Const_Iter: copy assignment
                                failed");
240                    return this;
241                }
242                self_reference operator++() { // preincrement
243                    if (curr_node->is_dummy)
```

```
244                        throw std::out_of_range("CDAL_Const_Iter: Can't traverse past the
                               end of the list");
245                    curr_array_index = (curr_array_index + 1) % array_size;
246                    if (curr_array_index == 0) curr_node = curr_node->next;
247                    return *this;
248                }
249            self_type operator++(int) { // postincrement
250                    self_type t(*this); //save state
251                    operator++(); //apply increment
252                    return t; //return state held before increment
253                }
254            reference operator*() const {
255                    return curr_node->item_array[curr_array_index];
256                }
257            pointer operator->() const {
258                    return & this->operator*();
259                }
260            bool operator==(const self_type& rhs) const {
261                    return rhs.curr_node == curr_node
262                            && rhs.curr_array_index == curr_array_index;
263                }
264            bool operator!=(const self_type& rhs) const {
265                    return ! operator==(rhs);
266                }
267        };

268

269        //---------------------------------------------------
270        // types
271        //---------------------------------------------------
272        typedef CDAL_Iter iterator;
273        typedef CDAL_Const_Iter const_iterator;
274        typedef T value_type;
275        //todo: might need to add size_t here and other iterators if they were
                excluded or commented out

276

277        iterator begin() {
278            ItemLoc start_loc = loc_from_pos(0);
279            return iterator(start_loc);
280        }

281

282        iterator end() {
283            ItemLoc end_loc = loc_from_pos(size());
284            return iterator(end_loc);
285        }

286

287        const_iterator begin() {
288            ItemLoc start_loc = loc_from_pos(0);
289            return const_iterator(start_loc);
290        }

291

292        const_iterator end() {
293            ItemLoc end_loc = loc_from_pos(size());
```

```cpp
                    return const_iterator(end_loc);
                }

                T& operator[](size_t i) {
                    if (i >= size()) {
                        throw std::out_of_range(std::string("operator[]: No element at
                            position ") + std::to_string(i));
                    }
                    return loc_from_pos(i).item_ref;
                }

                const T& operator[](size_t i) const {
                    if (i >= size()) {
                        throw std::out_of_range(std::string("operator[]: No element at
                            position ") + std::to_string(i));
                    }
                    return loc_from_pos(i).item_ref;
                }

                //---------------------------------------------------
                // Constructors/destructor/assignment operator
                //---------------------------------------------------

                CDAL() {
                    init();
                    embiggen_if_necessary();
                }
                //---------------------------------------------------
                //copy constructor
                CDAL(const CDAL& src) {
                    init();
                    copy_constructor(src);
                }

                //---------------------------------------------------
                //destructor
                ~CDAL() {
                    // safely dispose of this CDAL's contents
                    clear();
                }

                //---------------------------------------------------
                //copy assignment constructor
                CDAL& operator=(const CDAL& src) {
                    if (&src == this) // check for self-assignment
                        return *this;   // do nothing
                    // safely dispose of this CDAL's contents
                    // populate this CDAL with copies of the other CDAL's contents
                    clear();
                    init();
                    copy_constructor(src);
                    return *this;
```

```
        }

        //---------------------------------------------------
        // member functions
        //---------------------------------------------------

        /*
            replaces the existing element at the specified position with the
                specified element and
            returns the original element.
        */
        T replace(const T& element, size_t position) {
            T item = element;
            if (position >= size()) {
                throw std::out_of_range(std::string("replace: No element at position
                    ") + std::to_string(position));
            } else {
                ItemLoc loc = loc_from_pos(position);
                std::swap(loc.item_ref, item);
            }
            return item;
        }

        //---------------------------------------------------
        /*
            adds the specified element to the list at the specified position,
                shifting the element
            originally at that and those in subsequent positions one position to the
                right.
        */
        void insert(const T& element, size_t position) {
            if (position > size()) {
                throw std::out_of_range(std::string("insert: Position is outside of
                    the list: ") + std::to_string(position));
            } else {
                embiggen_if_necessary();
                ItemLoc loc = loc_from_pos(position);
                //shift remaining items to the right
                T item_to_insert = element;
                Node* n = loc.node;
                for (size_t i = position; i <= num_items; ++i) {
                    size_t array_index = i % array_size;
                    if ( i != position && array_index == 0 ) {
                        n = n->next;
                    }
                    std::swap(item_to_insert, n->item_array[array_index]);
                }
                ++num_items;
            }
        }

        //---------------------------------------------------
```

```
391         //Note to self: use reference here because we receive the original object
                instance,
392         //then copy it into n->item so we have it if the original element goes out
                of scope
393         /*
394             prepends the specified element to the list.
395         */
396         void push_front(const T& element) {
397             insert(element, 0);
398         }
399
400         //--------------------------------------------------
401         /*
402             appends the specified element to the list.
403         */
404         void push_back(const T& element) {
405             insert(element, size());
406         }
407
408         //--------------------------------------------------
409         //Note to self: no reference here, so we get our copy of the item, then
                return a copy
410         //of that so the client still has a valid instance if our destructor is
                called
411         /*
412             removes and returns the element at the list's head.
413         */
414         T pop_front() {
415             if (is_empty()) {
416                 throw std::out_of_range("pop_front: Can't pop: list is empty");
417             }
418             return remove(0);
419         }
420
421         //--------------------------------------------------
422         /*
423             removes and returns the element at the list's tail.
424         */
425         T pop_back() {
426             if (is_empty()) {
427                 throw std::out_of_range("pop_back: Can't pop: list is empty");
428             }
429             return remove(size() - 1);
430         }
431
432         //--------------------------------------------------
433         /*
434             removes and returns the the element at the specified position,
435             shifting the subsequent elements one position to the left.
436         */
437         T remove(size_t position) {
438             T old_item;
```

```cpp
            if (position >= size()) {
                throw std::out_of_range(std::string("remove: No element at position
                    ") + std::to_string(position));
            } else {
                ItemLoc loc = loc_from_pos(position);
                //shift remaining items to the left
                Node* n = loc.node;
                old_item = loc.item_ref;
                for (size_t i = position; i != num_items; ++i) {
                    size_t curr_array_index = i % array_size;
                    size_t next_array_index = (i + 1) % array_size;
                    T& curr_item = n->item_array[curr_array_index];
                    if ( next_array_index == 0 ) {
                        n = n->next;
                    }
                    T& next_item = n->item_array[next_array_index];
                    std::swap(curr_item, next_item);
                }
                --num_items;
                shrink_if_necessary();
            }
            return old_item;
        }

        //--------------------------------------------------
        /*
            returns (without removing from the list) the element at the specified
                position.
        */
        T item_at(size_t position) const {
            if (position >= size()) {
                throw std::out_of_range(std::string("item_at: No element at position
                    ") + std::to_string(position));
            }
            return operator[](position);
        }

        //--------------------------------------------------
        /*
            returns true IFF the list contains no elements.
        */
        bool is_empty() const {
            return size() == 0;
        }

        //--------------------------------------------------
        /*
            returns the number of elements in the list.
        */
        size_t size() const {
            return num_items;
        }
```

```
        //--------------------------------------------------
        /*
            removes all elements from the list.
        */
        void clear() {
            while (head->next != tail) {
                drop_node_after(head);
            }
            num_items = 0;
        }
        //--------------------------------------------------
        /*
            returns true IFF one of the elements of the list matches the specified
                element.
        */
        bool contains(const T& element,
                bool equals(const T& a, const T& b)) const {
            bool element_in_list = false;
            const_iterator fin = end();
            for (const_iterator iter = begin(); iter != fin; ++iter) {
                if (equals(*iter, element)) {
                    element_in_list = true;
                    break;
                }
            }
            return element_in_list;
        }

        //--------------------------------------------------
        /*
            If the list is empty, inserts "<empty list>" into the ostream;
            otherwise, inserts, enclosed in square brackets, the list's elements,
            separated by commas, in sequential order.
        */
        std::ostream& print(std::ostream& out) const {
            if (is_empty()) {
                out << "<empty list>";
            } else {
                out << "[";
                const_iterator start = begin();
                const_iterator fin = end();
                for (const_iterator iter = start; iter != fin; ++iter) {
                    if (iter != start)
                        out << ",";
                    out << *iter;
                }
                out << "]";
            }
            return out;
        }
    protected:
```

```cpp
        bool validate_internal_integrity() {
            //todo: fill this in
            return true;
        }
    }; //end class CDAL
} // end namespace cop3530
#endif // _CDAL_H_
```