

# 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()

## **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(size\_t num\_nodes\_to\_preallocate = 50)**

- Default constructor - takes a parameter which defines the initial array capacity

## **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`
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

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

- Copy assignment operator - starting from an arbitrary state, 1) reset to unialized 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`
- The resulting state of the current instance should be independent of the source (ie changing an element on the current instance should not affect the original)

## **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  $\geq 100$  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
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

### **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 true

## Const Iterator Methods

**explicit SDAL\_\_Const\_\_Iter(T\* item\_array, T\* end\_ptr)**

- 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
- When the client attempts to dereference an end iterator, the iterator should throw an out-of-range error

**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 true