COEN 79: Object-Oriented Programming
Homework #5

Tamir Enkhjargal

March 5, 2020

Question #1

What are the *iterator invalidation rules* of `std::list`?

For insertion: all iterators and references are unaffected.
For deletion: only iterators and references for erased element are invalidated.

Question #2

What are the *iterator invalidation rules* for STL's `vector` class?

For insertion: all iterators and references before the point of insertion are unaffected, unless the new block size is greater than before (in that case all iterators and references are invalidated).
For deletion: all iterators and references after point of deletion is invalidated.

Question #3

What are the features of a *random-access* iterator? Present the name of two STL data structures that offer random access iterators.

Random access iterators has all features of a bidirectional iterator: read and write, forward and backward moving. Random access iterators allow for access in any randomly selected location in a container.

Two STL data structures that have random access iterators are: vector and deque.

Question #4

Write the *pseudo-code* of an algorithm that evaluates a *fully parenthesized mathematical expression* using *stack* data structure (calculator).

```
float evaluate (string) {
    while(reading chars != EOF) {
        find number : push to number_stack
        find symbol : push to symbol_stack
        find right_parentheses : pop top 2 number_stack and top symbol and evaluate
    }
    return final_value
}
```

Question #5

The `node` class is defined as follows:

```
template <class Item>
class node {
public:
    // TYPEDEF
    typedef Item value_type;

    // CONSTRUCTOR
    node( const Item& init_data = Item(), node* init_link = NULL ) {
            data_field = init_data;
            link_field = init_link;
        }

    // MODIFICATION MEMBER FUNCTIONS
    Item& data() {  return data_field; }
    node* link() {  return link_field; }
    void set_data(const Item& new_data) { data_field = new_data;  }
    void set_link(node* new_link) {  link_field = new_link;  }

    // CONST MEMBER FUNCTIONS
    const Item& data() const {  return data_field;  }
    const node* link() const {  return link_field;  }

private:
    Item data_field;
    node* link_field;
};
```

Please write the implementation of a template ***const** forward iterator* for this class. Use *inline* functions in your implementation. The iterator is a *template class*.

```
template <class Item>
class const_node_iterator:public std::iterator<std::forward_iterator_tag, const Item> {
    private:
        const node<Item> *pointer;
    public:
        const_node_iterator(const node<Item> *init = NULL) { pointer = init; }
        const Item& operator*() { return pointer->data_field(); }
        const_node_iterator& operator++() { pointer = pointer->link();
                                        return *this; }
        const_node_iterator operator++(int) { node_iterator original(pointer);
                                        pointer = pointer->link_field();
                                        return original; }
}
```

Question #6

The bag class is defined as follows:

```
template < class Item >
class bag {
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef Item value_type;
    typedef std::size_t size_type;
    static const size_type DEFAULT_CAPACITY = 30;
    typedef bag_iterator < Item > iterator;

    bag(size_type initial_capacity = DEFAULT_CAPACITY);
    bag(const bag& source);
    ~bag();

    // MODIFICATION MEMBER FUNCTIONS
    // ...

    iterator begin();
    iterator end();

private:
    Item* data;          // Pointer to partially filled dynamic array
    size_type used;      // How much of array is being used
    size_type capacity;  // Current capacity of the bag
};
```

This class implements the following functions to create iterators:

```
template <class Item>
typename bag <Item>::iterator bag<Item>::begin() {
    return iterator(capacity, used, 0, data);
}

template <class Item>
typename bag<Item>::iterator bag<Item>::end() {
    return iterator(capacity, used, used, data);
}
```

Please complete the implementation of the following iterator:

```
template <class Item>
class bag_iterator::public std::iterator <std::forward_iterator_tag, Item> {
    public:
        iterator(size_type cap, size_type len, size_type cur, Item* init) {
            capacity = cap;
            used = len;
            current = cur;
            data = init; }
        Item& operator*() const { return *data; }
        iterator& operator++() { data++;
                                 return *this;}
        iterator operator++(int) { iterator original(data);
                                   data++;
                                   return original; }
        bool operator==(const iterator other) const {
                return (capacity == other.capacity && used == other.used &&
                        current == other.current &&
                        for(i = 0; i < count; i++) { data[i] == other.data[i] }}
        bool operator!=() {
                return (capacity != other.capacity || used != other.used ||
                        current != other.current ||
                        for(i = 0; i < count; i++) { data[i] != other.data[i] }}
    private:
        size_type capacity;
        size_type used;
        size_type current;
        Item* data;
}
```

Question #7

For the `queue` class given in Appendix 1 (cf. end of this assignment), implement the *copy constructor*. Note that the class uses a dynamic array. Also please note that you must not use the `copy` function (copy only the *valid entries* of one array to the new array). *The private member variables of the two objects must be exactly the same*.

```
template <class Item>
queue<Item>::queue(const queue <Item>& source) {
    for(i = 0; i < source.count; i++)
        data[i] = source.data[i];
    first = source.first;
    last = source.last;
    count = source.count;
    capacity = source.capacity;
}
```

Question #8

For the queue class given in Appendix 1 (cf. end of this assignment), implement the following function, which increases the size of the dynamic array used to store items. Please note that you must not use the copy function (copy only the valid entries of one array to the new array).

```
template <class Item>
void queue<Item>::reserve(size_type new_capacity) {
    value_type* larger_array;
    if(new_capacity == capacity) return;
    if(new_capacity < count) new_capacity = count;
    for(size_type i = 0; i < new_capacity; i++)
        larger_array[i] = data[i];
    delete data;
    capacity = new_capacity;
    data = larger_array;
}
```

Question #9

For the deque class given in Appendix 2 (cf. end of this assignment), implement the following constructor. The constructor allocates an array of block pointers and initializes all of its entries with NULL. The initial size of the array is init_bp_array_size.

```
template <class Item>
deque<Item>::deque(int init_bp_array_size, int init_block_size) {
    bp_array_size = init_bp_array_size;
    block_size = init_block_size;
    block_pointers = new value_type*[bp_array_size];
    for(size_type i = 0; i < bp_array_size; ++i)
        block_pointers[index] = NULL;
    block_pointers_end = block_pointers + bp_array_size - 1;
    first_bp = last_bp = front_ptr = back_ptr = NULL;
}
```

Question #10

For the deque class given in Appendix 2 (cf. end of this assignment), write the full implementation of the following function.

```
template <class Item>
void deque<Item>::pop_front() {
    assert(!isEmpty());
    if(back_ptr == front_ptr) {
        clear();
    }
```

```
    else if(front_ptr == ((*first_bp) + block_size - 1)) {
        delete[] *first_bp;
        *first_bp = NULL;
        ++first_bp;
        front_ptr = *first_bp;
    }
    else
        ++front_ptr;
}
```

Appendix 1:

`queue` `class` declaration:

```
template < class Item >
class queue {
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef std::size_t size_type;
    typedef Item value_type;

    static const size_type CAPACITY = 30;

    // CONSTRUCTOR and DESTRUCTOR
    queue(size_type initial_capacity = CAPACITY);
    queue(const queue& source);
    ~queue();

    // MODIFICATION MEMBER FUNCTIONS
    Item& front();
    void pop();
    void push(const Item & entry);
    void reserve(size_type new_capacity);

    // CONSTANT MEMBER FUNCTIONS
    bool empty() const {  return (count == 0);  }
    const Item & front() const;
    size_type size() const { return count;  }

private:
    Item* data;          // Circular array
    size_type first;     // Index of item at front of the queue
    size_type last;      // Index of item at rear of the queue
    size_type count;     // Total number of items in the queue
    size_type capacity;  // HELPER MEMBER FUNCTION

    size_type next_index(size_type i) const  { return (i + 1) % capacity; }
};
```

Appendix 2:

`deque class` declaration:

```
template < class Item >
class deque {
public:
    // TYPEDEF
    static const size_t BLOCK_SIZE = 5; // Number of data items per block

    // Number of entries in the block of array pointers. The minimum acceptable value is 2
    static const size_t BLOCKPOINTER_ARRAY_SIZE = 5;

    typedef std::size_t size_type;
    typedef Item value_type;

    deque(int init_bp_array_size = BLOCKPOINTER_ARRAY_SIZE,
                         int initi_block_size = BLOCK_SIZE);

    deque(const deque & source);
    ~deque();

    // CONST MEMBER FUNCTIONS
    bool isEmpty();
    value_type front();
    value_type back();

    // MODIFICATION MEMBER FUNCTIONS
    void operator = (const deque & source);
    void clear();
    void reserve();
    void push_front(const value_type & data);
    void push_back(const value_type & data);
    void pop_back();
    void pop_front();

private:
    // A pointer to the dynamic array of block pointers
    value_type** block_pointers;

    // A pointer to the final entry in the array of block pointers
    value_type** block_pointers_end;

    // A pointer to the first block pointer that's now being used
    value_type** first_bp;

    // A pointer to the last block pointer that's now being used
    value_type** last_bp;

    value_type* front_ptr; // A pointer to the front element of the whole deque
    value_type* back_ptr; // A pointer to the back element of the whole deque

    size_type bp_array_size; // Number of entries in the array of block pointers
    size_type block_size; // Number of entries in each block of items
};
```