Midterm Notes

Dynamic allocation of a pointer

```
double abc = 4.0;
double* x = &abc;
double** x1 = new double* (&abc);
std::cout << x << std::endl;
std::cout << *x1 << std::endl;
std::cout << **x1 << std::endl;
```

prints out:
0x7fff5fbff7c8
0x7fff5fbff7c8
4

Copy Constructors

- `ClassName(const ClassName &classVariable);`
- Default copy constructor copies all values verbatim.
    - This is a problem when it comes to dynamically allocated arrays
    - Both the original and new object's dynamically allocated arrays will point to the same location in memor

Assignment Operators

- `ClassName& operator=(const ClassName &classVariable);`
    - argument is a reference to the item on the right hand side
    - called via `object1 = object;`

Linked List Functions

```
~LinkedList()
    {
        LinkedListNode* iterator = m_head;
        while (iterator != nullptr){
            LinkedListNode* killme = iterator;
            iterator = iterator->next;
            delete killme;
        }
    }

// Project 2

//
//  Map.cpp
//  Project 2
//
//  Created by Bibek Ghimire on 1/20/16.
```

```
#include "Map.h"

//---------------------//
// MAP CLASS FUNCTIONS //
//---------------------//

// constructor
Map::Map() {
    m_head = nullptr;
    m_tail = nullptr;
    m_numPairs = 0;
}

// copy constructor
Map::Map(const Map& source) {
    m_numPairs = 0;

    // Required for insert() to loop properly
    m_head = nullptr;
    m_tail = nullptr;

    // Insert into new Map each pair from old map
    for (Node* a = source.m_tail; a != nullptr; a = a->prev) {
        insert(a->key, a->value);
    }
}

// destructor
Map::~Map() {
    Node* temp;

    // Delete each node in linked list
    for (Node* a = m_head; a != nullptr; a = temp) {
        temp = a->next;
        delete a;
    }
}

// assignment operator overload
Map& Map::operator=(const Map& source) {

    // If an object is being assigned to itself, return right away
    if (this == &source)
        return *this;

    Map temp(source);
    temp.swap(*this);
    return *this;
}

bool Map::empty() const {
    if (m_numPairs == 0) return true;
    else return false;
}

int Map::size() const {
    return m_numPairs;
}

bool Map::insert(const KeyType& key, const ValueType& value) {

    // If key is equal to any key currently in the map, cannot insert (should update instead)
```

```
    for (Node* a = m_head; a != nullptr; a = a->next)
        if (a->key == key)
            return false;

    // Create new node
    Node *p = new Node(key, value);
    m_numPairs++;

    // If the map is empty, simple insert
    if (m_head == nullptr)
        m_head = m_tail = p;
    // If map is not empty, add to top
    else {
        p->next = m_head;
        m_head->prev = p;
        m_head = p;
    }

    return true;
}

bool Map::update(const KeyType& key, const ValueType& value) {

    // Check all values in list
    for (Node* a = m_head; a != nullptr; a = a->next)
        // If the matching key is found, update its value
        if (a->key == key) {
            a->value = value;
            return true;
        }
    // Matching value was not found in list
    return false;
}

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value) {
    if (update(key, value))
        return true;
    if (insert(key, value))
        return true;
    return false;
}

bool Map::erase(const KeyType& key) {

    // Empty linked list, nothing to erase
    if (m_head == nullptr)
        return false;

    // One node in linked list
    if (m_numPairs == 1 && m_head->key == key) {
        delete m_head;                  // could also have deleted m_tail; it points to same
node
        m_head = nullptr;
        m_tail = nullptr;
        m_numPairs--;
        return true;
    }

    // Erase HEAD pointer
    if (m_head->key == key) {
        Node* temp = m_head;
        m_head = m_head->next;
        m_head->prev = nullptr;
        delete temp;
        m_numPairs--;
```

```
        return true;
    }

    // Erase TAIL pointer
    if (m_tail->key == key) {
        Node *temp = m_tail;
        m_tail = m_tail->prev;
        m_tail->next = nullptr;
        delete temp;
        m_numPairs--;
        return true;
    }

    // Node to erase is somewhere in the middle of the linked list
    for (Node* a = m_head; a != nullptr; a = a->next) {

        // Node to erase found
        if (a->key == key) {
            a->prev->next = a->next;
            a->next->prev = a->prev;
            delete a;
            m_numPairs--;
            return true;
        }
    }

    // Key does not exist
    return false;
}

bool Map::contains(const KeyType& key) const {

    // Check all values in list
    for (Node* a = m_head; a != nullptr; a = a->next)
        if (a->key == key)
            return true;

    return false;
}

bool Map::get(const KeyType& key, ValueType& value) const {

    // Empty list, no pairs to get
    if (m_numPairs == 0)
        return false;

    // Check all values in list
    for (Node* a = m_head; a != nullptr; a = a->next) {
        if (a->key == key) {
            value = a->value;
            return true;
        }
    }

    // Key not found
    return false;
}

bool Map::get(int i, KeyType& key, ValueType& value) const {

    // Empty list       or i out of range
    if (m_numPairs == 0 || i < 0 || i >= m_numPairs)
        return false;

    int count = 0;                      // Keep track of which node of linked list we look at
```

```
    for (Node* a = m_head; a != nullptr; a = a->next, count++) {
        if (count == i) {               // If the current node is the i-th node
            key = a->key;               //
            value = a->value;           // Send key-vaue pair to reference parameters
            return true;
        }
    }
    return false;
}

void Map::swap(Map& other) {

    // Save temporary variables
    Node* tempHead = this->m_head;
    Node* tempTail = this->m_tail;
    int tempNumPairs = this->m_numPairs;

    // Set this to other
    this->m_head = other.m_head;
    this->m_tail = other.m_tail;
    this->m_numPairs = other.m_numPairs;

    // Set other to this through temp
    other.m_head = tempHead;
    other.m_tail = tempTail;
    other.m_numPairs = tempNumPairs;

}

//----------------------//
// NON-MEMBER FUNCTIONS //
//----------------------//

bool combine(const Map& m1, const Map& m2, Map& result) {

    // Consider m1 and m2 input parameters, result output parameter

    // Input parameters are the same, output either one of them
    if (&m1 == &m2) {
        result = m1;
        return true;
    }

    Map temp;
    bool conflict = false;
    KeyType k1, k2;
    ValueType v1, v2;

    // Loop through m1's keys and values
    for (int i = 0; i < m1.size(); i++) {
        m1.get(i, k1, v1);

        // *** ADDS MATCHING KEY-VALUES *** /
        // Check if m2 has matching key
        if (m2.contains(k1)) {
            m2.get(k1, v2);

            // Check if m2's matching key has a matching value
            if (v1 == v2)
                temp.insert(k1, v1);

            // m2 has same key, different value, CONFLICT
            else
                conflict = true;
```

```
        // *** ADDS EXCLUSIVES FROM M1 *** /
        // Key from m1 not in m2, insert exclusive node
        } else
            temp.insert(k1, v1);
    }

    // *** ADDS EXCLUSIVES FROM M2 *** //
    for (int i = 0; i < m2.size(); i++) {
        m2.get(i, k2, v2);
        if (!m1.contains(k2))
            // been considered in above for loop
            temp.insert(k2, v2);
    }

    // Ensures result's incoming data doesn't affect output
    result = temp;
    return !conflict;
}

void subtract(const Map& m1, const Map& m2, Map& result) {
    Map temp;
    KeyType k;
    ValueType v;

    // Loop through m1
    for (int i = 0; i < m1.size(); i++) {
        m1.get(i, k, v);
        // Insert into result m1's exclusive pairs based on keys
        if (!m2.contains(k))
            temp.insert(k, v);
    }
    result = temp;
}
```