C++ Classes:

- Data members are initialized in the order they are declared in the class declaration
- Class objects will always call their default constructor first, unless initialized in a member initialization list

Constructors:

- A default constructor is assigned if no user-written constructor exists
    - Initializer lists can be used instead of initializing the member variables in the constructor's body
- Data members are initialized before the class object itself is created
    - Therefore, if a constructor creates an object of another class, it's constructor will run before the body of the current constructor does
- Watch out for if you need to use initializer lists!!!

Destructors:

- Often used to free dynamically allocated variables
- If you need a destructor, you likely need a copy constructor and assignment operator
- Objects that are created last are destructed first
- Destructors for an object will not be called unless the object itself is deleted, even if a pointer to the object passes out of scope

Copy Constructors:

- Copies an existing object into a new object
- Be careful of copying dynamically allocated data with the compiler's default copy constructor

Assignment Operator:

- Called when an existing object is assigned to another existing object
- Copy/swap for linked lists: ***Map m(n); swap(m); return *this;***
- Copy constructors and assignment operators both take parameters of ***const objectName& varName***
- Watch out for aliasing!!!

Linked Lists:

- Always test special cases: 0 element list, 1 element list, normal list
    - Test boundaries: head, tail, normal Node
- Conditions are tested left → right!!!
- Always ensure the arrow operator cannot be referencing a nullptr
- ***p->next = nullptr;*** - makes ptr point to the last Node

74

Linked List Cheat Sheet

```
struct Node
{
   string value;
   Node *next;
   Node  *prev;
};
```

Given a pointer to a node: Node *ptr;

NEVER access a node's data until validating its pointer:

```
if (ptr != nullptr)
    cout << ptr->value;
```

To advance ptr to the next node/end of the list:

```
if (ptr != nullptr)
    ptr = ptr->next;
```

To see if ptr points to the last node in a list:

```
if (ptr != nullptr && ptr->next == nullptr)
    then-ptr-points-to-last-node;
```

To get to the next node's data:

```
if (ptr != nullptr && ptr->next != nullptr)
    cout << ptr->next->value;
```

To get the head node's data:

```
if (head != nullptr)
    cout << head->value;
```

To check if a list is empty:

```
if (head == nullptr)
    cout << "List is empty";
```

Does our traversal meet this requirement?

```
NODE *ptr = head;
while (ptr != nullptr)
{
  cout << ptr->value;
  ptr = ptr->next;
}
```

To check if a pointer points to the first node in a list:

```
if (ptr == head)
    cout << "ptr is first node";
```

```
#include <iostream>
using namespace std;
class LinkedList
{
public:
    LinkedList(): head(nullptr) { }
    ~LinkedList();
    void addToList(int value);    // add to the end of the linked list
    void reverse();               // Reverse the linked list
    void output();
private:
    struct Node
    {
        int num;
        Node *next;
    };
    Node *head;
};
void LinkedList::output()
{
    Node *ptr = head;
    cout << "The elements in the list are: ";
    while(ptr!=nullptr) {
        cout << ptr->num << " ";
        ptr = ptr->next;
    }
    cout << endl;
}
int main()
{
    LinkedList list;
    for(int i=1;i<=10;i++)     list.addToList(i);
    list.output();
    list.reverse();
    list.output();
}
```

```
LinkedList::~LinkedList()
{
    Node *temp;
    while(head != nullptr) {
        temp = head;
        head = head->next;
        delete temp;
    }
}
```

```
void LinkedList::addToList(int value)
{
    Node *nodeToAdd = new Node;
    nodeToAdd->num = value;
    nodeToAdd->next = nullptr;
    if( head == nullptr)
        head = nodeToAdd;
    else {
        Node *ptr = head;
        while(ptr->next != nullptr)
            ptr = ptr->next;
        ptr->next = nodeToAdd;
    }
}
```

```
bool LinkedList::findNthFromLast(int N, int &value)
{
    if( N <= 0 ) return false;

    Node *ptr = head;
    int i, M = 0;
    // M is the number of nodes in the linked list.
    while(ptr != nullptr)
    {
        M++;
        ptr = ptr->next;
    }

    if( N > M ) return false;

    for( i=1, ptr = head; i < (M-N+1); i++)
        ptr = ptr->next;
    value = ptr->num;

    return true;
}
```

```
void LinkedList::reverse()
{
    Node *nextNode = nullptr,*prevNode = nullptr, *current = head;
    while(current) {
    // Hint: Only 4 lines of codes are needed inside the while loop
        nextNode = current->next;
        current->next = prevNode;
        prevNode = current;
        current = nextNode;
    }
    head = prevNode;
}
```

```
#include <iostream>
using namespace std;

class Triangle    {
public:
    Triangle() {
        p = new Point[3];
    }
    Triangle(int x1,int y1,int x2, int y2,int x3,int y3) {
        p = new Point[3];
        p[0].x = x1;    p[0].y = y1;
        p[1].x = x2;    p[1].y = y2;
        p[2].x = x3;    p[2].y = y3;
    }
    Triangle::~Triangle() {      delete [] p;       }
private:
    struct Point {
        int x,y;
        Point(int px=0,int py=0): x(px), y(py) { }
    };
    Point *p;
};

int main()    {
    Triangle *array[3];
    array[0] = new Triangle(1,1,1,3,3,1);
    array[1] = new Triangle(2,2,2,6,6,2);
    array[2] = new Triangle(3,3,3,9,9,3);
    Triangle c2 = *array[0];
    c2 = *array[1];

    for(int i=0;i<3;i++)
        delete array[i];
}
```

```
Triangle::Triangle(Triangle &t)
{
    p = new Point[3];

    for(int i=0;i<3;i++)
        p[i] = t.p[i];
}
```

```
Triangle& Triangle::operator=(const Triangle &t)
{
    if(&t == this)
        return *this;

    delete [] p;

    p = new Point[3];

    for(int i=0;i<3;i++)
        p[i] = t.p[i];

    return *this;
}
```

1. (a)

```
SortedLinkedList::SortedLinkedList()
{
    m_head = m_tail = nullptr;
    m_size = 0;
}
```

(b)

```
bool SortedLinkedList::insert(const ItemType& value)
{
    Node* p = m_head;
    Node* q = nullptr;

    while (p != nullptr)           // Find the first element with a greater value
    {                              // than the input value.
        if (value == p->m_value)
            return false;

        if (value < p->m_value)
            break;

        q = p;
        p = p->m_next;
    }

    Node* newNode = new Node;      // The new node must sit between q and p.
    newNode->m_value = value;
    newNode->m_next = p;
    newNode->m_prev = q;

    if (p != nullptr)              // Is there a following node?
        p->m_prev = newNode;
    else
        m_tail = newNode;

    if (q != nullptr)              // Is there a preceding node?
        q->m_next = newNode;
    else
        m_head = newNode;

    m_size++;
}
```

(c)

```
Node* SortedLinkedList::search(const ItemType& value) const
{
    for (Node* p = m_head; p != nullptr; p = p->m_next)
    {
        if (p->m_value == value)
            return p;
    }
        return nullptr;
}
```

(d)

```
void SortedLinkedList::remove(Node* node)
{
    if (node == nullptr)
        return;

    if (node != m_head)
        node->m_prev->m_next = node->m_next;
    else
        m_head = m_head->m_next;

    if (node != m_tail)
        node->m_next->m_prev = node->m_prev;
    else
        m_tail = m_tail->m_prev;

    delete node;
    m_size--;
}
```

(e)

```
void SortedLinkedList::printIncreasingOrder() const
{
    for (Node* p = m_head; p != nullptr; p = p->m_next)
        cout << p->m_value << endl;
}
```

Linked List Copy Constructor:

```
LL(const LL& other) {
    if(other.head == nullptr) {
        head = nullptr;
        tail = nullptr;
    else {
        head = new Node;
        head->val = other.head->val;
        head->next = other.head->next;
        head->prev = nullptr;
        Node* thisCurrent = head;
        Node* otherCurrent = other.head;
        while(otherCurrent->next != nullptr) {
            thisCurrent->next = new Node;
            thisCurrent->next->val = otherCurrent->next->val;
            thisCurrent->next->next = otherCurrent->next->next;
            thisCurrent->next->prev = thisCurrent;
            thisCurrent = thisCurrent->next;
            otherCurrent = otherCurrent->next;
        }
        tail = thisCurrent;
    }
```

```
LL(const LL& other) {
  if (other.head == nullptr)
    head = nullptr;
  else {
    head = new Node;
    head->val = other.head->val;
    head->next = other.head->next;

    Node* thisCurrent = head;
    Node* otherCurrent = other.head;
    while (otherCurrent->next != nullptr) {
      thisCurrent->next = new Node;
      thisCurrent->next->val = otherCurrent->next->val;
      thisCurrent->next->next = otherCurrent->next->next;

      thisCurrent = thisCurrent->next;
      otherCurrent = otherCurrent->next;
    }
  }
}
```