# Programming in C/C++: Supervision 2

Daniel Chatfield

December 1, 2014

1. Write an implementation of a class `LinkList` which stores zero or more positive integers internally as a linked list on the heap. The class should provide appropriate constructors and destructors and a method `pop()` to remove items from the head of the list. The method `pop()` should return `-1` if there are no remaining items. Your implementation should override the copy constructor and assignment operator to copy the linked-list structure between class instances. You might like to test your implementation with the following:

```
int main() {
    int test[] = {1,2,3,4,5};
    LinkList l1(test+1,4), l2(test,5);
    LinkList l3=l2, l4;
    l4=l1;
    printf("%d %d %d\n",l1.pop(),l3.pop(),l4.pop());
    return 0;
}
```

*Hint: heap allocation & deallocation should occur exactly once!*

```
#include <stdio.h>

class LinkedListItem {
public:
    int val;
    LinkedListItem *next;
    LinkedListItem(const int val, LinkedListItem* next=NULL);
};

class IllegalArgumentError {};

LinkedListItem::LinkedListItem(const int val, LinkedListItem* next) {
    if(val < 0) {
        throw IllegalArgumentError();// not sure how to handle this.
    }
    this->val = val;
    this->next = next;
}

class LinkedList {
    LinkedListItem* first;
public:
    LinkedList();
    virtual ~LinkedList();
    LinkedList(const LinkedList& other);
    LinkedList(const int vals[], const int length);
    LinkedList & operator=(const LinkedList& other);
    int pop();
    void print();
};
```

```
LinkedList::LinkedList() {
    this->first = NULL;
}


// Copy constructor
LinkedList::LinkedList(const LinkedList & other) {
    *this = other;
}

LinkedList::LinkedList(const int vals[], int length) {
    LinkedListItem *rv = NULL;
    for (; length > 0; length-- ) {
        rv = new LinkedListItem(vals[length - 1], rv);
    }
    this->first = rv;
}

LinkedList & LinkedList::operator=(const LinkedList & other) {
    LinkedListItem* otherCurrent = other.first;
    LinkedListItem* thisPrevious = NULL;
    while(otherCurrent != NULL) {
        LinkedListItem* thisCurrent = new LinkedListItem(otherCurrent->val, NULL
            );
        if (thisPrevious == NULL) {
            this->first = thisCurrent;
        } else {
            thisPrevious->next = thisCurrent;
        }

        thisPrevious = thisCurrent;
        otherCurrent = otherCurrent->next;
    }
    return *this;
}

int LinkedList::pop() {
    if(first == NULL) {
        return -1;
    }
    LinkedListItem *rv = this->first;
    this->first = this->first->next;
    return rv->val;
}

void LinkedList::print() {
    LinkedListItem* current = this->first;
    printf("Values {\n");
    while(current != NULL) {
        printf("    %d\n", current->val);
        current = current->next;
    }
    printf("}\n");
}


int main() {
    // Capital letters used in variable names for my sanity
    int test[] = {1,2,3,4,5};
    LinkedList L1(test+1,4), L2(test, 5);
    LinkedList L3 = L2, L4;
    L4 = L1;
    printf("%d %d %d\n", L1.pop(), L3.pop(), L4.pop());

    //outputs 2 1 2
}
```

2. If a function $f$ has a static instance of a class as a local variable, when might the class constructor be called?

3. Write a class `Matrix` which allows a programmer to define $2 \times 2$ matrices. Overload the common operators (e.g. $+$, $-$, $*$, and $/$). Can you easily extend your design to matrices of arbitrary size?

```
class Vector {
    float x;
    float y;

    friend class Matrix;

public:
    Vector(const float x, const float y);
    void pprint();
};

class Matrix {
    float a;
    float b;
    float c;
    float d;

public:

    Matrix(
        const float a,
        const float b,
        const float c,
        const float d
    );

    void pprint();

    Matrix operator+(const Matrix & right);
    Matrix operator-(const Matrix & right);
    Matrix operator*(const float x);
    Vector operator*(const Vector & right);
    Matrix operator*(const Matrix & right);
    Matrix operator/(const Matrix & right);
};
```

---

```
#include <stdio.h>
#include "q3.h"

Matrix::Matrix(
    const float a,
    const float b,
    const float c,
    const float d
    ) {
    this->a = a;
    this->b = b;
    this->c = c;
    this->d = d;
}
```

```
void Matrix::pprint() {
    printf("\n|%f␣%f|\n", this->a, this->b);
    printf("|%f␣%f|\n\n", this->c, this->d);
}

Matrix Matrix::operator+(const Matrix & right) {
    return Matrix(
        this->a + right.a,
        this->b + right.b,
        this->c + right.c,
        this->d + right.d
    );
}

Matrix Matrix::operator-(const Matrix & right) {
    return Matrix(
        this->a - right.a,
        this->b - right.b,
        this->c - right.c,
        this->d - right.d
    );
}

Matrix Matrix::operator*(const float x) {
    return Matrix(
    this->a * x,
    this->b * x,
    this->c * x,
    this->d * x
    );
}

Vector Matrix::operator*(const Vector & right) {
    return Vector(
        this->a * right.x + this->b * right.y,
        this->c * right.x + this->d * right.y
    );
}

Matrix Matrix::operator*(const Matrix & right) {
    return Matrix(
        this->a * right.a + this->b * right.c,
        this->a * right.b + this->b * right.d,
        this->c * right.a + this->d * right.c,
        this->c * right.b + this->d * right.d
    );
}

Matrix Matrix::operator/(const Matrix & right) {
    float inverseDeterminant
        = 1.0 / (right.a * right.d - right.b * right.c);

    return Matrix(
        this->a *  right.d + this->b * -right.c,
        this->a * -right.b + this->b *  right.a,
        this->c *  right.d + this->d * -right.c,
        this->c * -right.b + this->d *  right.a
    ) * inverseDeterminant;
}
```

This cannot easily be extended to matrices of arbitrary size as the arithmetic operations only work for matrices of certain sizes. For operations that are valid (e.g. multiplication of two $3 \times 3$ matrices) the implementation would have to be modified to store the elements as a 2D array.

4. Write a class `Vector` which allows a programmer to define a vector of length two. Modify your `Matrix` and `Vector` classes so that they interoperate correctly (e.g. `v2 = m*v1` should work as expected).

```
#include <stdio.h>
#include "q3.cpp"

Vector::Vector(const float x, const float y) {
    this->x = x;
    this->y = y;
}

void Vector::pprint(void) {
    printf("\n");
    printf("|%f|\n", this->x);
    printf("|%f|\n", this->y);
    printf("\n");
}

int main() {
    Matrix m(1,2,3,4);
    Matrix n(2,2,-1,0);
    m.pprint();
    n.pprint();

    Matrix o = m + n;

    o.pprint();

    Matrix p = m - n;

    p.pprint();

    Vector x = Vector(1,2);

    Vector y = m * x;

    y.pprint();
}
```

5. Why should destructors in an abstract class almost always be declared `virtual`?

> Otherwise when a class that inherits from it is casted back to the abstract class then the destructor defined in the abstract class will be called which may cause a memory leak if the inheriting class uses more memory.

6. Provide an implementation for: `template<class T> T Stack<T>::pop();` and `template<class T> Stack<T>::~Stack();` as declared in the slides for lecture 7.

> I initially wanted to do this by creating a destructor for the `Item` class but given it wasn't in the header file I figured we were meant to do it this way.

7. Provide an implementation for: `Stack(const Stack& s);` and

`Stack& operator=(const Stack& s);` as declared in the slides for lecture 7.

```
#include <cstdio>

template <class T> class Stack {

    struct Item {
        T val;
        Item* next;
        Item(T v) : val(v), next(0) {}
        Item(T v, Item* n) : val(v), next(n) {}
    };

    Item* head;

    Stack(const Stack& s);
    Stack& operator=(const Stack& s);

public:
    Stack() : head(0) {}
    ~Stack();
    T pop();
    void push(T val);
    void append(T val);
};

template<class T> void Stack<T>::append(T val) {
    Item **pp = &head;
    while(*pp) {
        pp = &((*pp)->next);
    }
    *pp = new Item(val);
}

int main() {
    Stack<char> s;
    s.push('a'), s.append('b'), s.pop();
}

class EmptyStackError{};

template<class T> void Stack<T>::push(T val) {
    this->head = new Item(val, this->head);
}

template<class T> T Stack<T>::pop() {
    if (this->head == NULL) {
        throw EmptyStackError();
    }
    Item *rv = this->head;
    this->head = this->head->next;
    return rv->val;
}

template<class T> Stack<T>::~Stack() {
    Item **pp = &this->head; // take a pointer to the head pointer
    while(*pp) { // while the pointer is still pointing to an item:
        Item &rv = **pp; // take a reference to the current item
        pp = &rv.next; //move pp onto the next one

        // delete current one (not sure whether need to explicitly delete the
            data)
        delete rv;
    }
}
```

```
template <class T> Stack<T>::Stack(const Stack& s) {

}
```

8. Using meta programming, write a templated class `Prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.

9. How can you be sure that your implementation of class `Prime` has been evaluated at compile time?