Solutions to Section Handout #4

1. Pointer tracing

2. Implementing a simple `IntVector` class

```cpp
/*
 * File: intvec.h
 * --------------
 * This interface file contains a simplified version of the IntVector
 * class in which the elements are always ints.
 */

#ifndef _INTVEC_H
#define _INTVEC_H

/**
 * Class: IntVector
 * -----------
 * This interface defines a simplified version of the IntVector
 * class in which the elements are always ints. This class supports
 * the basic functionality of the IntVector class defined in Chapter 4
 * with the exception of the following features:
 * - The class does not use templates to allow arbitrary element types
 * - Bracket selection is not supported
 * - Iterators and foreach are not supported
 * - No support is included for deep copying
 */

class IntVector {

public:

/*
 * Constructor: IntVector
 * Usage: IntVector vec;
 * --------------
 * The constructor initializes a new empty vector.
 */

    IntVector();

/*
 * Destructor: ~IntVector
 * --------------
 * Frees the storage associated with this vector.
 */

    ~IntVector();

*/
#endif
```
*/
* Method: size
* Usage: nElems = vec.size();
* ---------------------------------------------------------
* Returns the number of elements in this vector.
*/

int size();

/*
* Method: isEmpty
* Usage: if (vec.isEmpty())...
* ---------------------------------------------------------
* Returns true if this vector contains no elements, false otherwise.
*/

bool isEmpty();

/*
* Method: get
* Usage: val = vec.get(index);
* ---------------------------------------------------------
* Returns the element at the specified index in this vector.
* Elements are indexed starting with 0. Raises an error if
* index is outside of the vector bounds.
*/

int get(int index);

/*
* Method: set
* Usage: vec.set(index, value);
* ---------------------------------------------------------
* Replaces the element at the specified index in this vector with
* a new value. The previous value at that index is overwritten.
* Raises an error if index is outside of the vector bounds.
*/

void set(int index, int value);

/*
* Method: add
* Usage: vec.add(value);
* ---------------------------------------------------------
* Adds an element to the end of this vector.
*/

void add(int value);
/* Method: insert
* Usage: vec.insert(0, value);
*--------------------------------------------------
* Inserts the element into this vector before the specified index,
* shifting all subsequent elements one index higher. A call to
* vec.insert(0, val) inserts a new element at the beginning;
* vec.insert(vec.size(), val) adds a new element to the end.
* Raises an error if the index is out of range.
*/

void insert(int index, int value);

/*
* Method: remove
* Usage: vec.remove(3);
*------------------
* Removes the element at the specified index from this vector,
* shifting all subsequent elements one index lower. A call to
* vec.remove(0) removes the first element, while a call to
* vec.remove(vec.size()-1), removes the last. Raises an error
* if index is outside the range [0, size()-1].
*/

void remove(int index);

/*
* Method: clear
* Usage: vec.clear();
*---------------
* Removes all elements from this vector.
*/

void clear();

/* Private section */

private:

/* Instance variables */

int *elements; /* A dynamic array of the elements */
int capacity; /* The allocated size of the array */
int count; /* The number of elements in use */

/* Private method prototypes */

void expandCapacity();

};

#endif
/* 
 * File: intvec.cpp
 * ------------
 * This file contains the implementation of the intvec.h interface.
 * The implementation of most of these methods is so simple that no
 * detailed documentation is required.
 */

#include "error.h"
#include "intvec.h"
using namespace std;

/* Constants */

const int INITIAL_CAPACITY = 100;

/*
 * Implementation notes: IntVector constructor and destructor
 * -------------------------------------------------------------
 * The constructor allocates storage for the dynamic array
 * and initializes the other fields of the object. The
 * destructor frees the memory used for the array.
 */

IntVector::IntVector() {
    capacity = INITIAL_CAPACITY;
    count = 0;
    elements = new int[capacity];
}

IntVector::~IntVector() {
    delete[] elements;
}

int IntVector::size() {
    return count;
}

bool IntVector::isEmpty() {
    return count == 0;
}

int IntVector::get(int index) {
    if (index < 0 || index >= count) error("Index out of range");
    return elements[index];
}

void IntVector::set(int index, int elem) {
    if (index < 0 || index >= count) error("Index out of range");
    elements[index] = elem;
}
/ * Implementation notes: add, insert, remove
 * ----------------------------------------
 * These methods must shift the existing elements in the array to
 * make room for a new element or to close up the space left by a
 * deleted one.
 */

void IntVector::add(int elem) {
    insert(count, elem);
}

void IntVector::insert(int index, int elem) {
    if (index < 0 || index > count) error("Index out of range");
    if (count == capacity) expandCapacity();
    for (int i = count; i > index; i--) {
        elements[i] = elements[i - 1];
    }
    elements[index] = elem;
    count++;
}

void IntVector::remove(int index) {
    if (index < 0 || index >= count) error("Index out of range");
    for (int i = index; i < count - 1; i++) {
        elements[i] = elements[i + 1];
    }
    count--;
}

void IntVector::clear() {
    delete[] elements;
    capacity = INITIAL_CAPACITY;
    count = 0;
    elements = new int[capacity];
}

/*
 * Implementation notes: expandCapacity
 * ------------------------------
 * This private method doubles the capacity of the elements array
 * whenever it runs out of space. To do so, it must allocate a new
 * array, copy all the elements from the old array to the new one,
 * and free the old storage.
 */

void IntVector::expandCapacity() {
    capacity *= 2;
    int *oldElements = elements;
    elements = new int[capacity];
    for (int i = 0; i < count; i++) {
        elements[i] = oldElements[i];
    }
    delete[] oldElements;
}
3. Heap-stack diagrams

The following diagram uses explicit addresses, along with the internal numeric values of the Suit enumerated type:

This version eliminates addresses entirely and uses arrows to represent pointers. It also shows the Suit constant names instead of their underlying values:

Either format (or an intermediate hybrid) would be perfectly acceptable on an exam.
3b. explicit addresses

heap

stack

arrow representation

heap

stack
3c. explicit addresses

Arrow representation