Problem 1: Pointer tracing
What output is generated by the following program:

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

char *enigma(char *cp);
void mystery(char *cp);

int main() {
    char s1[4] = { 'H', 'H', 'H', '\0' };  
    char *s2 = enigma(s1); 
    *s2-- = '!';
    cout << s1 << endl;
    mystery(s2);
    *--s2 -= 5; 
    cout << s1 << endl;
    return 0;
}

char *enigma(char *cp) {
    char ch1 = *cp++;
    char ch2 = (*cp)++;
    if (ch1 == ch2) cp++;
    return cp;
}

void mystery(char str[]) {
    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = '+';
    }
}
```
**Problem 2: Implementing a simple IntVector class**

One of the most useful classes from Chapter 5 is the Vector class that provides for an improved version of an array. As it is presented in Chapter 5, the Vector class includes several special features, including the use of templates to support generic element types and square bracket notation to indicate selection. Although you’ll learn how to make those extensions later in the quarter, it is useful as an exercise to implement a simple version of Vector that works only with vectors of integers and that requires explicit method calls to `get` and `set` to implement selection. The methods supported by such a class—which you might call IntVector—are shown in Figure 1.

2a. Design an `intvec.h` interface that specifies these operations.

2b. Design a data structure that makes it possible to implement your `intvec.h` interface.

2c. Write a complete `intvec.cpp` file that implements the IntVector class.

For each of these tasks, the best model is the CharStack class in section 12.4 of the text.

---

**Figure 1 Methods exported by the IntVector class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IntVector()</code></td>
<td>Constructor for an empty IntVector.</td>
</tr>
<tr>
<td><code>~IntVector()</code></td>
<td>Destructor that frees any internal storage used for the vector.</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the number of elements in the vector.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns true if the vector is empty.</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>Returns the element of the vector that appears at the specified index position.</td>
</tr>
<tr>
<td><code>set(index, value)</code></td>
<td>Sets the element at the specified index to the new value. Attempting to reset an element outside the bounds of the vector generates an error.</td>
</tr>
<tr>
<td><code>add(value)</code></td>
<td>Adds a new element at the end of the vector.</td>
</tr>
<tr>
<td><code>insert(index, value)</code></td>
<td>Inserts the new value before the specified index position.</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>Deletes the element at the specified index position.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all elements from the vector.</td>
</tr>
</tbody>
</table>
3. Heap-stack diagrams

Using the heap-stack diagrams from Chapter 12 as a model, draw a diagram showing how memory is allocated for each of the following problems. You need not include explicit addresses in your diagram, but must indicate—either through addresses or arrows—where reference values point in memory. For extra practice, you might try drawing the heap-stack diagrams in both the explicit address and arrow forms.

3a. ```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES };

struct Card {
    int rank;
    Suit suit;
};

Card createCard(int rank, Suit suit);

int main() {
    Card *hand = new Card[2];
    hand[0] = createCard(9, DIAMONDS);
    hand[1] = createCard(7, HEARTS);
    return 0;
}

Card createCard(int rank, Suit suit) {
    Card card;
    card.rank = rank;
    card.suit = suit;
    return card;
} ←Diagram at this point on the second call to this function
```
3b. struct Student {
    int id;
    double gpa;
};

void initStudent(Student & s, int id);

int main() {
    Student advisees[2];
    advisees[0].id = 2718281;
    advisees->gpa = 4.00;
    initStudent(advisees[1], 3141592);
    return 0;
}

void initStudent(Student & s, int id) {
    Student *sptr = new Student;
    sptr->id = id;
    sptr->gpa = 3.33;
    s = *sptr;
} ←Diagram at this point just before the function returns

heap

stack
3c. class Matrix {
    public:
    
    Matrix(int size) {
        this->size = size;
        data = new double *[size];
        for (int i = 0; i < size; i++) {
            double *row = new double[size];
            data[i] = row;
            for (int j = 0; j < size; j++) {
                row[j] = (i == j) ? 1.0 : 0.0;
            }
        }
    }
    
    int getSize() {
        return size;
    }
    
    int get(int row, int col) {
        return data[row][col];
    }
    
    private:
    int size;
    double **data;
};

int main() {
    Matrix m(2);
    for (int i = 0; i < m.getSize(); i++) {
        for (int j = 0; j < m.getSize(); j++) {
            cout << " " << m.get(i, j);
        }
        cout << endl;
    }
    return 0;
}