

Object Oriented Programming – C++

Chapter 5: Pointers in C++

Coding Problems and Solutions — Set 3

This is a third, independent set of programming problems on this chapter, with no overlap with Sets 1 and 2. Each problem includes a complete, compilable C++ program, its expected output, and a short explanation where needed.

Problem 1: Find the frequency of each element in an integer array using pointers. [Medium]

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int arr[8] = {4, 2, 4, 5, 2, 4, 6, 5};
    int n = 8;
    int *p = arr;
    bool counted[8] = {false};
    for (int i = 0; i < n; i++) {
        if (counted[i]) continue;
        int count = 1;
        for (int j = i + 1; j < n; j++) {
            if (*(p + j) == *(p + i)) { count++; counted[j] = true; }
        }
        cout << *(p + i) << " occurs " << count << " time(s)" << endl;
    }
    return 0;
}
```

Output: 4 occurs 3 time(s)
2 occurs 2 time(s)
5 occurs 2 time(s)
6 occurs 1 time(s)

Problem 2: Merge two sorted integer arrays into a third array using pointers. [Medium]

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int a[4] = {1, 3, 5, 7};
    int b[3] = {2, 4, 6};
    int merged[7];
    int *pa = a, *pb = b, *pm = merged;
    int *aEnd = a + 4, *bEnd = b + 3;
    while (pa < aEnd && pb < bEnd) {
        if (*pa <= *pb) *pm++ = *pa++;
        else *pm++ = *pb++;
    }
    while (pa < aEnd) *pm++ = *pa++;
    while (pb < bEnd) *pm++ = *pb++;
    for (int i = 0; i < 7; i++) cout << merged[i] << " ";
}
```

```
    cout << endl;
    return 0;
}
```

Output: 1 2 3 4 5 6 7

Problem 3: Implement a queue using a dynamically allocated array with front and rear pointers (indices).

[Hard]

Solution:

```
#include <iostream>
using namespace std;
class Queue {
    int *arr;
    int capacity, front, rear, count;
public:
    Queue(int cap) : capacity(cap), front(0), rear(-1), count(0) {
        arr = new int[capacity];
    }
    void enqueue(int val) {
        if (count == capacity) { cout << "Queue full" << endl; return; }
        rear = (rear + 1) % capacity;
        arr[rear] = val;
        count++;
    }
    int dequeue() {
        if (count == 0) { cout << "Queue empty" << endl; return -1; }
        int val = arr[front];
        front = (front + 1) % capacity;
        count--;
        return val;
    }
    ~Queue() { delete[] arr; }
};
int main() {
    Queue q(3);
    q.enqueue(10); q.enqueue(20); q.enqueue(30);
    cout << q.dequeue() << endl;
    cout << q.dequeue() << endl;
    return 0;
}
```

Output: 10

20

Problem 4: Overload operator+ for a Complex class, using the 'this' pointer, to return a new object.

[Medium]

Solution:

```
#include <iostream>
using namespace std;
class Complex {
    float real, imag;
public:
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}
    Complex operator+(const Complex &other) {
```

```

        return Complex(this->real + other.real, this->imag + other.imag);
    }
    void display() { cout << real << " + " << imag << "i" << endl; }
};
int main() {
    Complex c1(2, 3), c2(4, 5);
    Complex c3 = c1 + c2;    // internally: c1.operator+(c2)
    c3.display();
    return 0;
}

```

Output: 6 + 8i

Explanation: Inside operator+, this points to c1 (the object on the left of the + sign), so this->real and this->imag refer to c1's data, while other refers to c2, the argument passed on the right.

Problem 5: Implement a doubly linked list: insert at the end and print the list forward. [Hard]

Solution:

```

#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *prev;
    Node *next;
};
void insertEnd(Node *&head, int val) {
    Node *n = new Node{val, nullptr, nullptr};
    if (!head) { head = n; return; }
    Node *t = head;
    while (t->next) t = t->next;
    t->next = n;
    n->prev = t;
}
void printForward(Node *head) {
    while (head) { cout << head->data << " <-> "; head = head->next; }
    cout << "NULL" << endl;
}
int main() {
    Node *head = nullptr;
    insertEnd(head, 5); insertEnd(head, 10); insertEnd(head, 15);
    printForward(head);
    return 0;
}

```

Output: 5 <-> 10 <-> 15 <-> NULL

Explanation: Each node now stores two pointers, next and prev, allowing traversal in both directions — this is what distinguishes a doubly linked list from the singly linked list seen earlier.

Problem 6: Search for a substring within a string using pointers (your own version of strstr). [Hard]

Solution:

```

#include <iostream>
using namespace std;
char* findSubstring(char *text, char *pattern) {
    for (char *t = text; *t != '\0'; t++) {

```

```

        char *t1 = t, *p1 = pattern;
        while (*p1 != '\0' && *t1 == *p1) { t1++; p1++; }
        if (*p1 == '\0') return t;    // full pattern matched
    }
    return nullptr;
}
int main() {
    char text[] = "programming in C++";
    char pattern[] = "in C";
    char *result = findSubstring(text, pattern);
    if (result) cout << "Found at: " << result << endl;
    else cout << "Not found" << endl;
    return 0;
}

```

Output: Found at: in C++

Explanation: The function returns a pointer into the middle of text, pointing to where the match begins; printing a char* prints the string starting from that address to the end.

Problem 7: Multiply two matrices using dynamically allocated 2D arrays (pointer-to-pointer). [Hard]

Solution:

```

#include <iostream>
using namespace std;
int main() {
    int r1 = 2, c1 = 2, r2 = 2, c2 = 2;
    int **A = new int*[r1];
    int **B = new int*[r2];
    int **C = new int*[r1];
    for (int i = 0; i < r1; i++) A[i] = new int[c1];
    for (int i = 0; i < r2; i++) B[i] = new int[c2];
    for (int i = 0; i < r1; i++) C[i] = new int[c2]{0};

    int va = 1, vb = 1;
    for (int i = 0; i < r1; i++) for (int j = 0; j < c1; j++) A[i][j] = va++;
    for (int i = 0; i < r2; i++) for (int j = 0; j < c2; j++) B[i][j] = vb++;

    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c2; j++)
            for (int k = 0; k < c1; k++)
                C[i][j] += A[i][k] * B[k][j];

    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) cout << C[i][j] << " ";
        cout << endl;
    }
    for (int i = 0; i < r1; i++) delete[] A[i];
    for (int i = 0; i < r2; i++) delete[] B[i];
    for (int i = 0; i < r1; i++) delete[] C[i];
    delete[] A; delete[] B; delete[] C;
    return 0;
}

```

Output: 7 10
15 22

Problem 8: Find the Student with the highest marks in an object array using pointer arithmetic (no subscripts). [Medium]

Solution:

```
#include <iostream>
using namespace std;
class Student {
public:
    string name;
    int marks;
};
int main() {
    Student list[4] = { {"Ravi", 72}, {"Meena", 91}, {"Ajay", 85}, {"Sneha", 88} };
    Student *p = list;
    Student *best = p;
    for (Student *q = p; q < p + 4; q++) {
        if (q->marks > best->marks) best = q;
    }
    cout << "Topper: " << best->name << " with " << best->marks << " marks" << endl;
    return 0;
}
```

Output: *Topper: Meena with 91 marks*

Problem 9: Employee hierarchy: base-class pointer array calling a virtual display() on Manager and Engineer objects. [Hard]

Solution:

```
#include <iostream>
using namespace std;
class Employee {
protected:
    string name;
public:
    Employee(string n) : name(n) {}
    virtual void display() { cout << name << " - Employee" << endl; }
    virtual ~Employee() {}
};
class Manager : public Employee {
public:
    Manager(string n) : Employee(n) {}
    void display() override { cout << name << " - Manager" << endl; }
};
class Engineer : public Employee {
public:
    Engineer(string n) : Employee(n) {}
    void display() override { cout << name << " - Engineer" << endl; }
};
int main() {
    Employee *staff[2];
    staff[0] = new Manager("Kabir");
    staff[1] = new Engineer("Divya");
    for (int i = 0; i < 2; i++) staff[i]->display();
    for (int i = 0; i < 2; i++) delete staff[i];
    return 0;
}
```

Output: Kabir - Manager
Divya - Engineer

Problem 10: Allocate a 2D array using a single pointer (linear allocation) with manual row/column indexing. [Hard]

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int rows = 2, cols = 3;
    int *matrix = new int[rows * cols];    // single contiguous block

    int val = 1;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            matrix[i * cols + j] = val++;    // manual 2D indexing

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            cout << matrix[i * cols + j] << " ";
        cout << endl;
    }
    delete[] matrix;    // single delete, since it is one block
    return 0;
}
```

Output: 1 2 3
4 5 6

Explanation: Unlike the pointer-to-pointer (`int**`) approach, this allocates one contiguous block of `rows*cols` integers, so element (i, j) is found at offset $i*cols + j$. It needs only a single `delete[]`, and is generally more cache-friendly than an array of separately allocated rows.

Problem 11: Perform binary search on a sorted array using pointers, returning a pointer to the found element (or `nullptr`). [Medium]

Solution:

```
#include <iostream>
using namespace std;
int* binarySearch(int *arr, int n, int key) {
    int *low = arr, *high = arr + n - 1;
    while (low <= high) {
        int *mid = low + (high - low) / 2;
        if (*mid == key) return mid;
        else if (*mid < key) low = mid + 1;
        else high = mid - 1;
    }
    return nullptr;
}
int main() {
    int arr[7] = {2, 5, 8, 12, 16, 23, 38};
    int *result = binarySearch(arr, 7, 16);
    if (result) cout << "Found at index " << (result - arr) << endl;
    else cout << "Not found" << endl;
}
```

```
    return 0;
}
```

Output: Found at index 4

Problem 12: Write a generic swap function using void pointers and a byte-by-byte copy. [Hard]

Solution:

```
#include <iostream>
#include <cstring>
using namespace std;
void genericSwap(void *a, void *b, size_t size) {
    char *temp = new char[size];
    memcpy(temp, a, size);
    memcpy(a, b, size);
    memcpy(b, temp, size);
    delete[] temp;
}
int main() {
    int x = 10, y = 20;
    genericSwap(&x, &y, sizeof(int));
    cout << "x = " << x << ", y = " << y << endl;

    double p = 1.5, q = 2.5;
    genericSwap(&p, &q, sizeof(double));
    cout << "p = " << p << ", q = " << q << endl;
    return 0;
}
```

Output: x = 20, y = 10
p = 2.5, q = 1.5

Explanation: A void* pointer can hold the address of any data type but cannot be dereferenced directly, since the compiler does not know its size. Here, the function instead copies size raw bytes at a time using memcpy(), which makes the same swap logic work for any data type.

Problem 13: Implement a dynamically resizing stack: when full, allocate a bigger array, copy elements, and free the old array. [Hard]

Solution:

```
#include <iostream>
using namespace std;
class Stack {
    int *arr;
    int capacity, top;
    void resize() {
        int newCapacity = capacity * 2;
        int *newArr = new int[newCapacity];
        for (int i = 0; i <= top; i++) newArr[i] = arr[i];
        delete[] arr; // free the old, smaller block
        arr = newArr;
        capacity = newCapacity;
    }
public:
    Stack(int cap) : capacity(cap), top(-1) { arr = new int[capacity]; }
    void push(int val) {
```

```

        if (top == capacity - 1) resize();
        arr[++top] = val;
    }
    void print() { for (int i = 0; i <= top; i++) cout << arr[i] << " "; cout << endl; }
    ~Stack() { delete[] arr; }
};
int main() {
    Stack s(2);
    s.push(1); s.push(2); s.push(3);    // triggers resize on the 3rd push
    s.print();
    return 0;
}

```

Output: 1 2 3

Problem 14: Implement a binary search tree: insert nodes and print an inorder traversal, using pointers.

[Hard]

Solution:

```

#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *left;
    Node *right;
};
Node* insert(Node *root, int val) {
    if (root == nullptr) return new Node{val, nullptr, nullptr};
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}
void inorder(Node *root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
int main() {
    Node *root = nullptr;
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    for (int v : values) root = insert(root, v);
    inorder(root);
    cout << endl;
    return 0;
}

```

Output: 20 30 40 50 60 70 80

Explanation: Each Node holds left and right pointers to its two children, forming a tree instead of a linear list. insert() is a recursive function that returns a Node* so that the parent's left or right pointer can be updated with the (possibly newly created) subtree root.

Problem 15: Demonstrate that an array decays into a pointer when passed to a function, by comparing sizeof(arr) in main() with sizeof(arr) inside the function. [Medium]

Solution:

```
#include <iostream>
using namespace std;
void checkSize(int arr[]) {
    // Inside the function, arr is really just a pointer
    cout << "Inside function, sizeof(arr) = " << sizeof(arr) << " bytes" << endl;
}
int main() {
    int arr[10];
    cout << "Inside main, sizeof(arr) = " << sizeof(arr) << " bytes" << endl;
    checkSize(arr);
    return 0;
}
```

Output: Inside main, sizeof(arr) = 40 bytes

Inside function, sizeof(arr) = 8 bytes (size of a pointer)

Explanation: Inside main(), arr is the actual array of 10 ints (10 x 4 = 40 bytes). Once it is passed to a function, it decays into a plain int* pointer to its first element, so sizeof(arr) inside the function reports only the size of a pointer (commonly 8 bytes on a 64-bit system), not the size of the whole array — this is why array size must always be passed separately as an extra parameter.