

Object Oriented Programming – C++

Chapter 5: Pointers in C++

Coding Problems and Solutions — Set 2

This document gives complete, compilable solutions to the Set 2 practice questions, with expected output and short explanations where needed.

Problem 1: Declare a pointer to an array of 5 integers and use it to access all elements. [Medium]

Declare a pointer of type `int (*)[5]` and use it to print all the elements of an integer array of size 5.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int (*p)[5] = &arr;    // pointer to an array of 5 ints
    for (int i = 0; i < 5; i++) {
        cout << (*p)[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output: 10 20 30 40 50

Explanation: `p` is a pointer to the whole array (its type is 'array of 5 ints'), not to a single `int`, so it must first be dereferenced with `(*p)` to get back the array, and then indexed normally.

Problem 2: Array of character pointers, each pointing to a different string. [Medium]

Solution:

```
#include <iostream>
using namespace std;
int main() {
    const char *names[3] = {"Amit", "Priya", "Rahul"};
    for (int i = 0; i < 3; i++) {
        cout << names[i] << endl;
    }
    return 0;
}
```

Output: Amit

Priya

Rahul

Explanation: `names` is an array of pointers; each element stores the address of the first character of a separate string literal stored elsewhere in memory.

Problem 3: Find the index of a given element using pointer arithmetic. [Medium]

Solution:

```

#include <iostream>
using namespace std;
int findIndex(int *arr, int n, int key) {
    for (int *p = arr; p < arr + n; p++) {
        if (*p == key) return p - arr;    // pointer subtraction
    }
    return -1;
}
int main() {
    int arr[6] = {4, 9, 15, 23, 8, 16};
    cout << "Index of 23 = " << findIndex(arr, 6, 23) << endl;
    cout << "Index of 99 = " << findIndex(arr, 6, 99) << endl;
    return 0;
}

```

Output: Index of 23 = 3

Index of 99 = -1

Explanation: Subtracting two pointers of the same type ($p - arr$) gives the number of elements between them, which is exactly the index of the matched element.

Problem 4: Count the number of words in a sentence using a character pointer. [Medium]

Solution:

```

#include <iostream>
using namespace std;
int main() {
    char text[] = "Pointers are powerful in C++";
    char *p = text;
    int words = 0;
    bool inWord = false;
    while (*p != '\0') {
        if (*p != ' ' && !inWord) { words++; inWord = true; }
        else if (*p == ' ') inWord = false;
        p++;
    }
    cout << "Number of words = " << words << endl;
    return 0;
}

```

Output: Number of words = 5

Problem 5: Copy one character array into another using pointers only. [Easy]

Solution:

```

#include <iostream>
using namespace std;
void copyString(char *dest, const char *src) {
    while (*src != '\0') {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
}
int main() {

```

```

char source[] = "pointer copy";
char destination[50];
copyString(destination, source);
cout << destination << endl;
return 0;
}

```

Output: *pointer copy*

Problem 6: Convert Celsius to Fahrenheit and Kelvin in one call using reference parameters. [Medium]

Solution:

```

#include <iostream>
using namespace std;
void convertTemperature(float celsius, float *fahrenheit, float *kelvin) {
    *fahrenheit = (celsius * 9 / 5) + 32;
    *kelvin = celsius + 273.15f;
}
int main() {
    float c = 37, f, k;
    convertTemperature(c, &f, &k);
    cout << "Fahrenheit = " << f << endl;
    cout << "Kelvin = " << k << endl;
    return 0;
}

```

Output: *Fahrenheit = 98.6*
Kelvin = 310.15

Problem 7: Add two dynamically allocated arrays element-wise. [Medium]

Solution:

```

#include <iostream>
using namespace std;
int main() {
    int n = 4;
    int *a = new int[n]{1, 2, 3, 4};
    int *b = new int[n]{10, 20, 30, 40};
    int *result = new int[n];
    for (int i = 0; i < n; i++) result[i] = a[i] + b[i];
    for (int i = 0; i < n; i++) cout << result[i] << " ";
    cout << endl;
    delete[] a; delete[] b; delete[] result;
    return 0;
}

```

Output: *11 22 33 44*

Problem 8: Dynamically allocate a 2D matrix using pointer-to-pointer. [Hard]

Solution:

```

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

```

```

for (int i = 0; i < rows; i++) matrix[i] = new int[cols];

int value = 1;
for (int i = 0; i < rows; i++)
    for (int j = 0; j < cols; j++)
        matrix[i][j] = value++;

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) cout << matrix[i][j] << " ";
    cout << endl;
}

for (int i = 0; i < rows; i++) delete[] matrix[i];    // delete rows first
delete[] matrix;                                     // then the row-pointer array
return 0;
}

```

Output: 1 2 3
4 5 6

Explanation: *matrix is a pointer to pointers: matrix[i] itself is a separately allocated row. Deallocation must happen in reverse order — each row first, then the array of row pointers — otherwise the row pointers would be lost before they could be freed, causing a memory leak.*

Problem 9: Attempt to modify a value through a pointer to a constant integer. [Medium]

Solution:

```

#include <iostream>
using namespace std;
int main() {
    int x = 10;
    const int *p = &x;
    // *p = 20;    // COMPILE ERROR: assignment of read-only location '*p'
    cout << *p << endl;    // reading is allowed
    return 0;
}

```

Output: 10

Explanation: *const int *p means 'p points to a constant int', so the value at that address cannot be modified through p (though p itself can be made to point elsewhere, and x can still be modified directly, just not via p).*

Problem 10: Array of Student objects created dynamically; compute average marks through a pointer. [Medium]

Solution:

```

#include <iostream>
using namespace std;
class Student {
public:
    int roll;
    float marks;
};
int main() {
    int n = 3;
    Student *arr = new Student[n]{ {1, 78}, {2, 85}, {3, 91} };
}

```

```

float total = 0;
for (int i = 0; i < n; i++) total += arr[i].marks;
cout << "Average marks = " << total / n << endl;
delete[] arr;
return 0;
}

```

Output: Average marks = 84.6667

Problem 11: Overload == for a Complex class using the 'this' pointer. [Hard]

Solution:

```

#include <iostream>
using namespace std;
class Complex {
    float real, imag;
public:
    Complex(float r, float i) : real(r), imag(i) {}
    bool operator==(const Complex &other) {
        return (this->real == other.real) && (this->imag == other.imag);
    }
};
int main() {
    Complex c1(3, 4), c2(3, 4), c3(1, 2);
    cout << boolalpha;
    cout << "c1 == c2: " << (c1 == c2) << endl;
    cout << "c1 == c3: " << (c1 == c3) << endl;
    return 0;
}

```

Output: c1 == c2: true
c1 == c3: false

Problem 12: DynamicArray class with a deep-copy copy constructor and destructor. [Hard]

This solution covers both the copy constructor (deep copy) and the destructor together, since they are closely related.

Solution:

```

#include <iostream>
using namespace std;
class DynamicArray {
    int *data;
    int size;
public:
    DynamicArray(int n) : size(n) { data = new int[size]; }
    // Deep copy constructor
    DynamicArray(const DynamicArray &other) : size(other.size) {
        data = new int[size]; // fresh memory, NOT the same pointer
        for (int i = 0; i < size; i++) data[i] = other.data[i];
    }
    void set(int i, int val) { data[i] = val; }
    void print() { for (int i = 0; i < size; i++) cout << data[i] << " "; cout << endl; }
    ~DynamicArray() { delete[] data; } // release owned memory
};
int main() {

```

```

DynamicArray a(3);
a.set(0, 1); a.set(1, 2); a.set(2, 3);
DynamicArray b = a;    // calls deep-copy constructor
b.set(0, 99);          // modifying b must not affect a
a.print();
b.print();
return 0;
}

```

Output: 1 2 3
99 2 3

Explanation: If the default (shallow) copy constructor were used instead, both *a.data* and *b.data* would point to the very same block of memory. Modifying *b* would then also change *a*, and when both objects were destroyed, *delete[]* would be called twice on the same pointer, causing undefined behaviour (a double-free error).

Problem 13: Abstract base class Shape (pure virtual area()) with Square and Triangle, accessed through an array of base pointers. [Hard]

Solution:

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual float area() = 0;    // pure virtual -> Shape is abstract
    virtual ~Shape() {}
};
class Square : public Shape {
    float side;
public:
    Square(float s) : side(s) {}
    float area() override { return side * side; }
};
class Triangle : public Shape {
    float base, height;
public:
    Triangle(float b, float h) : base(b), height(h) {}
    float area() override { return 0.5f * base * height; }
};
int main() {
    Shape *shapes[2];
    shapes[0] = new Square(4);
    shapes[1] = new Triangle(6, 5);
    for (int i = 0; i < 2; i++) cout << "Area = " << shapes[i]->area() << endl;
    for (int i = 0; i < 2; i++) delete shapes[i];
    return 0;
}

```

Output: Area = 16
Area = 15

Problem 14: Base class with a virtual destructor; delete a Derived object through a Base pointer. [Hard]

Solution:

```

#include <iostream>
using namespace std;

```

```

class Base {
public:
    virtual ~Base() { cout << "Base destructor" << endl; }
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived destructor" << endl; }
};
int main() {
    Base *b = new Derived();
    delete b;
    return 0;
}

```

Output: *Derived destructor*
Base destructor

Explanation: Because `~Base()` is virtual, `delete b` correctly calls the most-derived destructor first (*Derived*), which then automatically invokes the base destructor. Both parts of the object are cleaned up properly.

Problem 15: Same scenario WITHOUT a virtual destructor — showing incomplete cleanup. [Hard]

Solution:

```

#include <iostream>
using namespace std;
class Base {
public:
    ~Base() { cout << "Base destructor" << endl; }    // NOT virtual
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived destructor" << endl; }
};
int main() {
    Base *b = new Derived();
    delete b;
    return 0;
}

```

Output: *Base destructor*

Explanation: Without the keyword `virtual`, `delete b` is resolved at compile time using the pointer's declared type (*Base*), so only `~Base()` is called. `~Derived()` never runs, which can leave any resources owned specifically by the *Derived* part of the object unreleased — this is why a base class intended to be used polymorphically should always declare a virtual destructor.

Problem 16: Frequency count of digits using `calloc()`. [Medium]

Solution:

```

#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    int number = 552031;
    int *freq = (int*) calloc(10, sizeof(int));    // zero-initialised
    int n = number;
}

```

```

while (n != 0) {
    int digit = n % 10;
    freq[digit]++;
    n /= 10;
}
for (int i = 0; i < 10; i++)
    if (freq[i] > 0) cout << i << " -> " << freq[i] << endl;
free(freq);
return 0;
}

```

Output: 0 -> 1

1 -> 1

2 -> 1

3 -> 1

5 -> 2

Problem 17: Array-based stack using dynamically allocated memory (push/pop). [Hard]

Solution:

```

#include <iostream>
using namespace std;
class Stack {
    int *arr;
    int capacity, top;
public:
    Stack(int cap) : capacity(cap), top(-1) { arr = new int[capacity]; }
    void push(int val) {
        if (top == capacity - 1) { cout << "Stack full" << endl; return; }
        arr[++top] = val;
    }
    int pop() {
        if (top == -1) { cout << "Stack empty" << endl; return -1; }
        return arr[top--];
    }
    ~Stack() { delete[] arr; }
};
int main() {
    Stack s(3);
    s.push(10); s.push(20); s.push(30);
    cout << s.pop() << endl;
    cout << s.pop() << endl;
    return 0;
}

```

Output: 30

20

Problem 18: Delete a specific node (by value) from a singly linked list. [Hard]

Solution:

```

#include <iostream>
using namespace std;
struct Node { int data; Node *next; };
void insertEnd(Node *&head, int val) {
    Node *n = new Node{val, nullptr};
}

```

```

    if (!head) { head = n; return; }
    Node *t = head;
    while (t->next) t = t->next;
    t->next = n;
}
void deleteValue(Node *&head, int val) {
    Node *cur = head, *prev = nullptr;
    while (cur != nullptr && cur->data != val) { prev = cur; cur = cur->next; }
    if (cur == nullptr) return; // value not found
    if (prev == nullptr) head = cur->next; // deleting the head node
    else prev->next = cur->next;
    delete cur;
}
void printList(Node *head) {
    while (head) { cout << head->data << " -> "; head = head->next; }
    cout << "NULL" << endl;
}
int main() {
    Node *head = nullptr;
    insertEnd(head, 10); insertEnd(head, 20); insertEnd(head, 30);
    deleteValue(head, 20);
    printList(head);
    return 0;
}

```

Output: 10 -> 30 -> NULL

Problem 19: Demonstrate a memory leak in a loop, then fix it. [Medium]

Solution (leaking version):

```

for (int i = 0; i < 5; i++) {
    int *p = new int(i); // allocated every iteration
    // missing: delete p; -> memory leaks each time
}

```

Corrected version:

```

#include <iostream>
using namespace std;
int main() {
    for (int i = 0; i < 5; i++) {
        int *p = new int(i);
        cout << *p << " ";
        delete p; // freed every iteration, no leak
    }
    cout << endl;
    return 0;
}

```

Output: 0 1 2 3 4

Problem 20: Dynamic array of Book objects with new[] / delete[]. [Medium]

Solution:

```

#include <iostream>
using namespace std;
class Book {

```

```
public:
    string title;
    float price;
};
int main() {
    int n = 3;
    Book *books = new Book[n]{
        {"C++ Basics", 250},
        {"Data Structures", 320},
        {"OOP Concepts", 280}
    };
    float total = 0;
    for (int i = 0; i < n; i++) total += books[i].price;
    cout << "Total price = " << total << endl;
    delete[] books;
    return 0;
}
```

Output: *Total price = 850*