

Object Oriented Programming – C++

Chapter 5: Pointers in C++

Coding Problems and Solutions

This set contains programming problems based on pointer concepts, pointers with functions and objects, and dynamic memory management, arranged in increasing order of difficulty. Each problem includes a complete, compilable C++ program, its expected output, and a short explanation where needed.

Problem 1: Swap two numbers using pointers (call by reference). *[Easy]*

Write a program that swaps the values of two integer variables by passing their addresses to a function.

Solution:

```
#include <iostream>
using namespace std;
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 9;
    swap(&x, &y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Output: x = 9, y = 5

Problem 2: Find the sum and average of array elements using pointer arithmetic only. *[Easy]*

Write a program that computes the sum and average of the elements of an integer array without using array subscript ([]) notation anywhere.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int arr[6] = {12, 7, 19, 4, 22, 6};
    int n = 6;
    int *p = arr;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(p + i);
    }
    cout << "Sum = " << sum << endl;
    cout << "Average = " << (float)sum / n << endl;
    return 0;
}
```

Output: Sum = 70
Average = 11.6667

Problem 3: Reverse a C-style string using pointers. [Easy]

Write a program that reverses a character array in place, using two pointers moving towards each other from opposite ends.

Solution:

```
#include <iostream>
#include <cstring>
using namespace std;
void reverseString(char *str) {
    char *start = str;
    char *end = str + strlen(str) - 1;
    while (start < end) {
        char temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}
int main() {
    char text[] = "pointer";
    reverseString(text);
    cout << text << endl;
    return 0;
}
```

Output: *retniop*

Problem 4: Count vowels in a string using a pointer. [Easy]

Write a program that counts the number of vowels in a string by walking through it with a character pointer.

Solution:

```
#include <iostream>
#include <cctype>
using namespace std;
int main() {
    char text[] = "Object Oriented Programming";
    char *p = text;
    int count = 0;
    while (*p != '\0') {
        char c = tolower(*p);
        if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u') count++;
        p++;
    }
    cout << "Number of vowels = " << count << endl;
    return 0;
}
```

Output: *Number of vowels = 9*

Problem 5: Add two numbers using a pointer to a pointer (double indirection). [Medium]

Write a program that stores two numbers using single pointers, then accesses both values through a pointer-to-pointer to compute and display their sum.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int a = 15, b = 25;
    int *pa = &a, *pb = &b;
    int **ppa = &pa, **ppb = &pb;
    int sum = **ppa + **ppb;
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Output: Sum = 40

Problem 6: Dynamically allocate an array with new, find its maximum, then release it with delete[].

[Medium]

Write a program that reads n integers into a dynamically allocated array, finds the maximum value, and correctly frees the memory afterwards.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int n = 5;
    int *arr = new int[n];
    int values[] = {23, 45, 12, 67, 34};
    for (int i = 0; i < n; i++) arr[i] = values[i];
    int maxVal = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxVal) maxVal = arr[i];
    cout << "Maximum = " << maxVal << endl;
    delete[] arr;    // release array memory
    return 0;
}
```

Output: Maximum = 67

Problem 7: Create a class object dynamically with new, use a pointer to invoke its member functions, then release it with delete. [Medium]

Write a program for a Rectangle class with length and breadth, where the object is created dynamically and its area is computed and displayed through an object pointer.

Solution:

```
#include <iostream>
using namespace std;
class Rectangle {
    int length, breadth;
public:
    Rectangle(int l, int b) : length(l), breadth(b) {}
    int area() { return length * breadth; }
};
int main() {
    Rectangle *r = new Rectangle(8, 5);    // dynamic object
```

```

    cout << "Area = " << r->area() << endl;
    delete r;                               // release memory
    return 0;
}

```

Output: Area = 40

Problem 8: Use the 'this' pointer to compare two objects of the same class for equality. [Medium]

Write a program for a Point class with x and y coordinates, where a member function isEqual() uses the 'this' pointer to compare the calling object with another object passed to it.

Solution:

```

#include <iostream>
using namespace std;
class Point {
    int x, y;
public:
    Point(int x, int y) { this->x = x; this->y = y; }
    bool isEqual(Point &other) {
        return (this->x == other.x) && (this->y == other.y);
    }
};
int main() {
    Point p1(3, 4), p2(3, 4), p3(1, 2);
    cout << boolalpha;
    cout << "p1 == p2: " << p1.isEqual(p2) << endl;
    cout << "p1 == p3: " << p1.isEqual(p3) << endl;
    return 0;
}

```

Output: p1 == p2: true
p1 == p3: false

Problem 9: Demonstrate runtime polymorphism using a base-class pointer to derived-class objects (Shape hierarchy). [Medium]

Write a program with a base class Shape having a virtual function area(), and derived classes Circle and Rectangle that override it. Use a Shape pointer to call area() on objects of both derived classes.

Solution:

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual float area() { return 0; }
};
class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    float area() override { return 3.14159f * radius * radius; }
};
class Rectangle : public Shape {
    float len, wid;
public:

```

```

    Rectangle(float l, float w) : len(l), wid(w) {}
    float area() override { return len * wid; }
};
int main() {
    Shape *s;
    Circle c(5);
    Rectangle r(4, 6);
    s = &c; cout << "Circle area = " << s->area() << endl;
    s = &r; cout << "Rectangle area = " << s->area() << endl;
    return 0;
}

```

Output: Circle area = 78.5397
 Rectangle area = 24

Problem 10: Store an array of base-class pointers pointing to different derived objects and call a virtual function on each (polymorphic array). [Hard]

Extend the previous Shape hierarchy so that an array of Shape pointers holds a mix of Circle and Rectangle objects, and their areas are printed in a single loop.

Solution:

```

#include <iostream>
using namespace std;
class Shape {
public:
    virtual float area() { return 0; }
    virtual ~Shape() {}
};
class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    float area() override { return 3.14159f * radius * radius; }
};
class Rectangle : public Shape {
    float len, wid;
public:
    Rectangle(float l, float w) : len(l), wid(w) {}
    float area() override { return len * wid; }
};
int main() {
    Shape *shapes[3];
    shapes[0] = new Circle(3);
    shapes[1] = new Rectangle(4, 5);
    shapes[2] = new Circle(2);
    for (int i = 0; i < 3; i++)
        cout << "Area " << i + 1 << " = " << shapes[i]->area() << endl;
    for (int i = 0; i < 3; i++) delete shapes[i]; // clean up
    return 0;
}

```

Output: Area 1 = 28.2743
 Area 2 = 20
 Area 3 = 12.5664

Explanation: Each element of the `shapes[]` array is a *Shape* pointer, but because `area()` is declared virtual, the call `shapes[i]->area()` is dispatched at run time to the correct override for whichever derived object each pointer actually refers to.

Problem 11: Manage a dynamic array using `malloc()`, `calloc()`, and `free()`. [Medium]

Write a program that allocates one array with `malloc()` and another of the same size with `calloc()`, prints both before any values are assigned to show the difference in initial contents, then frees both.

Solution:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    int n = 4;
    int *a = (int*) malloc(n * sizeof(int));    // uninitialised
    int *b = (int*) calloc(n, sizeof(int));     // zero-initialised
    cout << "calloc array (before assignment): ";
    for (int i = 0; i < n; i++) cout << b[i] << " ";
    cout << endl;
    for (int i = 0; i < n; i++) a[i] = (i + 1) * 10;
    cout << "malloc array (after assignment): ";
    for (int i = 0; i < n; i++) cout << a[i] << " ";
    cout << endl;
    free(a);
    free(b);
    return 0;
}
```

Output: *calloc array (before assignment): 0 0 0 0*
malloc array (after assignment): 10 20 30 40

Problem 12: Access a data member through a pointer-to-member operator (`.*` and `->*`). [Hard]

Write a program that declares a pointer to a data member of a class, and accesses that member both through an object (using `.*`) and through a pointer to the object (using `->*`).

Solution:

```
#include <iostream>
using namespace std;
class Account {
public:
    double balance = 5000.0;
};
int main() {
    Account acc;
    Account *pAcc = &acc;
    double Account::*pBalance = &Account::balance;

    cout << "Via object (.*): " << acc.*pBalance << endl;
    cout << "Via pointer (->*): " << pAcc->*pBalance << endl;
    return 0;
}
```

Output: Via object (. *): 5000
Via pointer (-> *): 5000

Problem 13: Implement a singly linked list and insert nodes at the end using self-referential structure pointers. [Hard]

Write a program that defines a Node structure containing a data field and a pointer to the next Node, then inserts several nodes at the end of the list and prints the list.

Solution:

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *next;
};
void insertEnd(Node *&head, int value) {
    Node *newNode = new Node{value, nullptr};
    if (head == nullptr) { head = newNode; return; }
    Node *temp = head;
    while (temp->next != nullptr) temp = temp->next;
    temp->next = newNode;
}
void printList(Node *head) {
    Node *temp = head;
    while (temp != nullptr) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}
int main() {
    Node *head = nullptr;
    insertEnd(head, 10);
    insertEnd(head, 20);
    insertEnd(head, 30);
    printList(head);
    return 0;
}
```

Output: 10 -> 20 -> 30 -> NULL

Explanation: Each Node holds a pointer, next, to another Node of the same type — this is called a self-referential structure. head is passed by reference (Node *&head) so that the function can update the caller's head pointer when the very first node is inserted.

Problem 14: Demonstrate a dangling pointer and show how to avoid it. [Medium]

Write a program that first shows what happens when a pointer is used after its memory has been freed (a dangling pointer), and then shows the safe practice of resetting the pointer to nullptr immediately after delete.

Solution:

```
#include <iostream>
using namespace std;
int main() {
    int *p = new int(99);
```

```

    cout << "Before delete: " << *p << endl;

    delete p;          // memory released
    p = nullptr;       // safe practice: avoid dangling pointer

    if (p == nullptr) {
        cout << "Pointer safely reset to nullptr after delete." << endl;
    } else {
        cout << *p;     // unsafe: would be undefined behaviour
    }
    return 0;
}

```

Output: Before delete: 99

Pointer safely reset to nullptr after delete.

Explanation: Had *p* not been reset, dereferencing it after delete would read memory that no longer belongs to this object, producing undefined behaviour. Checking for *nullptr* before use is a standard safeguard against dangling pointers.

Problem 15: Write a function that dynamically allocates memory and returns a pointer to it. [Hard]

Write a function `createArray(int n)` that dynamically allocates an array of *n* integers, fills it with the squares of 1..*n*, and returns the pointer to the caller, which prints and then frees it.

Solution:

```

#include <iostream>
using namespace std;
int* createArray(int n) {
    int *arr = new int[n];
    for (int i = 0; i < n; i++) arr[i] = (i + 1) * (i + 1);
    return arr;          // returning pointer to dynamic memory
}
int main() {
    int n = 5;
    int *result = createArray(n);
    for (int i = 0; i < n; i++) cout << result[i] << " ";
    cout << endl;
    delete[] result;     // caller is responsible for freeing it
    return 0;
}

```

Output: 1 4 9 16 25

Explanation: It is safe to return a pointer to memory allocated with *new*, because that memory lives on the heap and is not destroyed when the function returns (unlike a pointer to a local stack variable, which would become dangling). The responsibility for calling `delete[]` then passes to the caller.