PROGRAMMING IN C

Chapter 4: Functions

Practice Problems with Solutions

This set of problems covers all the topics of the chapter — need for functions, prototype declaration, scope and lifetime of variables, function definition, parameter passing, call by value and call by reference, return values, storage classes, categories of functions, and recursion. Each problem is followed by a complete, working C program and a step-by-step explanation of its logic and output.

Section A: Functions — Definition, Declaration and Calling

Problem 1: Simple function with no argument and no return value

Write a C program using a function that displays a welcome message on the screen. The function should neither accept any argument nor return any value.

Solution:

```
#include <stdio.h>

void welcome(void);        /* function prototype */

int main(void)
{
    welcome();              /* function call */
    return 0;
}

void welcome(void)          /* function definition */
{
    printf("Welcome to the C Programming Lab!\n");
}
```

Output:

```
Welcome to the C Programming Lab!
```

Explanation:

The prototype void welcome(void); tells the compiler in advance that a function named welcome exists, accepts no arguments, and returns nothing. main() calls this function, which transfers control to its definition; the printf() statement executes, and control then returns to main(). This is an example of the 'no argument, no return value' category of function.

Problem 2: Function with no argument but a return value

Write a C program with a function that reads an integer entered by the user and returns it to main(), where it is printed after doubling.

Solution:

```
#include <stdio.h>

int readNumber(void);

int main(void)
```

```
{
    int n = readNumber();   /* receives value, sends none */
    printf("Double of entered number = %d\n", n * 2);
    return 0;
}

int readNumber(void)
{
    int x;
    printf("Enter an integer: ");
    scanf("%d", &x);
    return x;
}
```

Output:

```
Enter an integer: 15
Double of entered number = 30
```

Explanation:

readNumber() takes no parameters, but its return type is int, so it sends the value typed by the user back to main() through the return statement. main() stores this value in n and then performs the doubling and printing itself, illustrating the 'no argument, with return value' category.

Problem 3: Function with argument and return value

Write a C program with a function that accepts the radius of a circle as an argument and returns its area.

Solution:

```
#include <stdio.h>
#define PI 3.14159

float area(float r);

int main(void)
{
    float radius, result;
    printf("Enter radius: ");
    scanf("%f", &radius);
    result = area(radius);      /* argument passed, value returned */
    printf("Area of circle = %.2f\n", result);
    return 0;
}

float area(float r)
{
    return PI * r * r;
}
```

Output:

```
Enter radius: 4
Area of circle = 50.27
```

Explanation:

The actual parameter radius in main() is copied into the formal parameter r of area(). The function computes PI * r * r and sends the result back using return, demonstrating the 'argument with return value' category, the most commonly used form of function.

Section B: Parameter Passing — Call by Value and Call by Reference

Problem 4: Demonstrating call by value

Write a C program to show that a function using call by value cannot change the original value of a variable declared in main().

Solution:

```
#include <stdio.h>

void addTen(int x);

int main(void)
{
    int a = 20;
    printf("Before function call: a = %d\n", a);
    addTen(a);                  /* a copy of a is passed */
    printf("After function call:  a = %d\n", a);
    return 0;
}

void addTen(int x)
{
    x = x + 10;                 /* changes only the local copy */
    printf("Inside function:     x = %d\n", x);
}
```

Output:

```
Before function call: a = 20
Inside function:     x = 30
After function call:  a = 20
```

Explanation:

When addTen(a) is called, the value of a (20) is copied into the formal parameter x. The statement x = x + 10 only modifies this local copy, which exists in its own memory location. Once the function returns, x is destroyed, and a in main() remains unchanged at 20 — the defining characteristic of call by value.

Problem 5: Demonstrating call by reference using pointers

Write a C program that uses call by reference (pointers) to swap the values of two variables.

Solution:

```
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
    int a = 5, b = 9;
    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);               /* addresses passed */
    printf("After swap:  a = %d, b = %d\n", a, b);
    return 0;
}

void swap(int *p, int *q)
{
    int temp = *p;
```

```
    *p = *q;
    *q = temp;
}
```

Output:

```
Before swap: a = 5, b = 9
After swap:  a = 9, b = 5
```

Explanation:

Instead of passing the values of a and b, their addresses (&a and &b) are passed to swap(). The formal parameters p and q are pointers that store these addresses, so *p and *q refer directly to a and b in main(). Any change made through *p or *q therefore changes the original variables — this is how call by reference is implemented in C.

> ***Note:*** *A function that needs to return more than one value to the caller (as in this swap example) must use call by reference, since a return statement can send back only a single value.*

Section C: Scope, Lifetime and Storage Classes

Problem 6: Local versus global variables

Write a C program to illustrate the difference between a local variable and a global variable with the same name.

Solution:

```
#include <stdio.h>

int num = 100;              /* global variable */

void display(void);

int main(void)
{
    int num = 5;             /* local variable, hides global num inside main */
    printf("Local num in main = %d\n", num);
    display();
    return 0;
}

void display(void)
{
    printf("Inside display, global num = %d\n", num);
}
```

Output:

```
Local num in main = 5
Inside display, global num = 100
```

Explanation:

Inside main(), declaring a local variable num creates a new variable whose scope is limited to main(); it hides (shadows) the global num for the duration of main() only. The function display(), however, has no local variable named num, so it refers directly to the global num, which still holds the value 100. This shows that scope, not the order of declaration alone, decides which variable a given statement accesses.

Problem 7: Effect of the static storage class

Write a C program with a function that counts how many times it has been called, using a static variable, and call it three times from main().

Solution:

```
#include <stdio.h>

void counter(void);

int main(void)
{
    counter();
    counter();
    counter();
    return 0;
}

void counter(void)
{
    static int count = 0;    /* initialized only on the first call */
    count++;
    printf("counter() has been called %d time(s)\n", count);
}
```

Output:

```
counter() has been called 1 time(s)
counter() has been called 2 time(s)
counter() has been called 3 time(s)
```

Explanation:

Because count is declared static, it is initialized to 0 only once, the first time counter() is called, and it is not destroyed when the function returns. On each subsequent call, count retains the value left over from the previous call, so it correctly accumulates the number of calls. If count had been an ordinary (automatic) local variable instead, it would be reset to 0 at the start of every call, and the output would show 1 each time.

Problem 8: Using the extern storage class across functions in the same file

Write a C program in which one function sets the value of a global variable and another function, declared using extern, uses that value.

Solution:

```
#include <stdio.h>

int total;                /* global definition */

void setTotal(int value);
void showTotal(void);

int main(void)
{
    setTotal(250);
    showTotal();
    return 0;
}

void setTotal(int value)
{
    total = value;
}
```

```
void showTotal(void)
{
    extern int total;       /* refers to the global 'total' above */
    printf("Total = %d\n", total);
}
```

Output:
```
Total = 250
```

Explanation:

total is defined once at the global level. The extern int total; declaration inside showTotal() does not create a new variable; it simply informs the compiler that total already exists elsewhere (here, at file/global scope), so showTotal() can access the same memory location that setTotal() modified. In multi-file programs, extern is used the same way to share a global variable that is actually defined in a different .c file.

Section D: Recursion

Problem 9: Factorial of a number using recursion

Write a recursive C function to compute the factorial of a given non-negative integer n.

Solution:

```
#include <stdio.h>

long factorial(int n);

int main(void)
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld\n", n, factorial(n));
    return 0;
}

long factorial(int n)
{
    if (n == 0)             /* base case */
        return 1;
    else
        return n * factorial(n - 1);   /* recursive case */
}
```

Output:
```
Enter a non-negative integer: 5
Factorial of 5 = 120
```

Explanation:

factorial(5) calls factorial(4), which calls factorial(3), and so on, until factorial(0) is reached. Since 0 satisfies the base case, it returns 1 immediately without any further calls. Each pending multiplication is then completed as the calls return in reverse order: factorial(1)=1, factorial(2)=2, factorial(3)=6, factorial(4)=24, and finally factorial(5)=120. Each call creates its own stack frame holding its own copy of n, and these frames are popped off the call stack one by one as the functions return.

Problem 10: Sum of natural numbers using recursion (linear recursion)

Write a recursive function to find the sum of the first n natural numbers.

Solution:

```
#include <stdio.h>

int sum(int n);

int main(void)
{
    int n;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("Sum of first %d natural numbers = %d\n", n, sum(n));
    return 0;
}

int sum(int n)
{
    if (n == 0)
        return 0;             /* base case */
    return n + sum(n - 1);    /* one recursive call: linear recursion */
}
```

Output:

```
Enter n: 4
Sum of first 4 natural numbers = 10
```

Explanation:

Each call to sum() makes exactly one further recursive call to itself, which is the defining feature of linear recursion. sum(4) = 4 + sum(3) = 4 + 3 + sum(2) = ... = 4 + 3 + 2 + 1 + sum(0), and sum(0) returns 0 directly. Adding these up gives 4 + 3 + 2 + 1 + 0 = 10.

Problem 11: Fibonacci numbers using recursion (tree recursion)

Write a recursive C function to print the first n terms of the Fibonacci series.

Solution:

```
#include <stdio.h>

int fib(int n);

int main(void)
{
    int n, i;
    printf("Enter number of terms: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        printf("%d ", fib(i));
    printf("\n");
    return 0;
}

int fib(int n)
{
    if (n == 0 || n == 1)     /* base cases */
        return n;
```

```
    return fib(n - 1) + fib(n - 2);   /* two recursive calls: tree recursion */
}
```

Output:

```
Enter number of terms: 7
0 1 1 2 3 5 8
```

Explanation:

Each call to fib(n), for n greater than 1, makes two further recursive calls — fib(n - 1) and fib(n - 2) — so the calls branch out like a tree rather than forming a single chain; this is called tree (or multiple) recursion. The two base cases, fib(0) = 0 and fib(1) = 1, stop the branching. Although simple to write, this approach recomputes the same smaller Fibonacci values many times, making it far less efficient than an iterative version for large n.

Problem 12: Checking whether a string is a palindrome using recursion

Write a recursive C function to check whether a given string is a palindrome (reads the same forwards and backwards).

Solution:

```
#include <stdio.h>
#include <string.h>

int isPalindrome(char str[], int start, int end);

int main(void)
{
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    if (isPalindrome(str, 0, strlen(str) - 1))
        printf("%s is a palindrome.\n", str);
    else
        printf("%s is not a palindrome.\n", str);
    return 0;
}

int isPalindrome(char str[], int start, int end)
{
    if (start >= end)          /* base case: crossed or met in the middle */
        return 1;
    if (str[start] != str[end])
        return 0;              /* mismatch found */
    return isPalindrome(str, start + 1, end - 1);  /* tail recursion */
}
```

Output:

```
Enter a string: madam
madam is a palindrome.
```

Explanation:

The function compares the characters at the start and end positions of the string and moves inward on each recursive call. If any pair of characters does not match, it returns 0 immediately. If start becomes greater than or equal to end, every pair has matched, so it returns 1. Since the recursive call is the last action performed with no pending operation after it returns, this is an example of tail recursion.

Section E: Applied Problems

Problem 13: Function returning the greatest of three numbers, called multiple times

Write a C program with a function greatest(int a, int b, int c) that returns the largest of three integers. Use it to compare two different sets of numbers entered by the user, demonstrating reusability of a function.

Solution:

```
#include <stdio.h>

int greatest(int a, int b, int c);

int main(void)
{
    int a1, b1, c1, a2, b2, c2;

    printf("Enter first set of three numbers: ");
    scanf("%d %d %d", &a1, &b1, &c1);
    printf("Greatest of first set = %d\n", greatest(a1, b1, c1));

    printf("Enter second set of three numbers: ");
    scanf("%d %d %d", &a2, &b2, &c2);
    printf("Greatest of second set = %d\n", greatest(a2, b2, c2));

    return 0;
}

int greatest(int a, int b, int c)
{
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}
```

Output:

```
Enter first set of three numbers: 12 45 9
Greatest of first set = 45
Enter second set of three numbers: 80 21 63
Greatest of second set = 80
```

Explanation:

The same function greatest() is called twice from main() with two completely different sets of actual parameters. Each call creates its own independent set of formal parameters and the local variable max, so the two calls do not interfere with each other. This program illustrates the reusability benefit of functions discussed as one of the main reasons for their use.

Problem 14: Function that both takes an argument and modifies a variable by reference

Write a C program with a function that accepts an array and its size (call by reference for the array), and returns the sum of its elements (return value), while also using a pointer parameter to return the maximum element found.

Solution:

```
#include <stdio.h>

int sumAndMax(int arr[], int n, int *maxOut);
```

```
int main(void)
{
    int arr[] = {12, 45, 3, 67, 29};
    int n = 5, maximum, total;

    total = sumAndMax(arr, n, &maximum);
    printf("Sum = %d\n", total);
    printf("Maximum = %d\n", maximum);
    return 0;
}

int sumAndMax(int arr[], int n, int *maxOut)
{
    int i, total = 0;
    *maxOut = arr[0];
    for (i = 0; i < n; i++)
    {
        total += arr[i];
        if (arr[i] > *maxOut)
            *maxOut = arr[i];
    }
    return total;              /* one value returned directly */
}
```

Output:

```
Sum = 156
Maximum = 67
```

Explanation:

An array name in C is passed to a function as the address of its first element, so arr[] inside sumAndMax() refers to the same memory as the original array — a natural example of call by reference. The function directly returns the sum using return, but since it needs to send back a second value (the maximum) as well, it uses an additional pointer parameter maxOut; the statement *maxOut = ... writes the result into main()'s maximum variable through this pointer. This problem combines both parameter-passing techniques discussed in this chapter.

Problem 15: Comparing an iterative and a recursive solution

Write two versions of a function to compute the power x^n (x raised to n, where n is a non-negative integer) — one using a simple loop (iterative) and one using recursion. Compare their logic.

Solution:

```
#include <stdio.h>

int powerIterative(int x, int n);
int powerRecursive(int x, int n);

int main(void)
{
    int x = 3, n = 4;
    printf("%d^%d (iterative) = %d\n", x, n, powerIterative(x, n));
    printf("%d^%d (recursive) = %d\n", x, n, powerRecursive(x, n));
    return 0;
}

int powerIterative(int x, int n)
{
```

```
    int result = 1, i;
    for (i = 0; i < n; i++)
        result *= x;
    return result;
}

int powerRecursive(int x, int n)
{
    if (n == 0)                    /* base case */
        return 1;
    return x * powerRecursive(x, n - 1);   /* recursive case */
}
```

Output:

```
3^4 (iterative) = 81
3^4 (recursive) = 81
```

Explanation:

Both functions produce the same result. powerIterative() uses a for loop and a single set of variables that are updated repeatedly, with no extra function calls or stack frames created. powerRecursive() instead breaks the problem into x multiplied by a smaller instance of the same problem, powerRecursive(x, n - 1), until the base case n == 0 is reached; this creates one stack frame per recursive call. This comparison highlights that recursion is often more elegant to write for problems with a naturally repetitive/self-similar structure, but an iterative solution generally uses less memory since it does not build up multiple stack frames.