Class 06

Topic: Recursion, String, Pointer

Recursion

Recursion means a function calling itself to solve smaller subproblems until it reaches a simple condition known as the base case.
Each recursive call goes deeper until the base case is met, and then the results start returning back — this process is known as *unwinding*.

Structure of a recursive function:

```
returnType functionName(parameters)
{
    if(base condition)
        return value;     // base case
    else
        return something + functionName(smaller parameter);  // recursive
case
}
```

Example 1: Factorial using recursion

```
#include<stdio.h>

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

int main()
{
```

```
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Factorial = %d", factorial(num));
    return 0;
}
```

Explanation:

When you call `factorial(5)`, it keeps calling `factorial(4)`, then `factorial(3)`, and so on until `factorial(0)` returns 1. Then it multiplies back:

`5 * 4 * 3 * 2 * 1 = 120`.

Example 2: Fibonacci using recursion

```
#include<stdio.h>

int fibonacci(int n)
{
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
    int n;
    printf("Enter number of terms: ");
    scanf("%d", &n);

    for(int i = 0; i < n; i++)
        printf("%d ", fibonacci(i));
    return 0;
}
```

Explanation:

Each Fibonacci number is the sum of the previous two. For example, `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`, and those again call smaller terms until reaching base cases `0` and `1`.

Important points about recursion

1. Every recursion must have a base case. Without it, the program will never stop and cause *stack overflow*.
2. Each recursive call takes memory on the call stack.
3. When the base case is met, control starts returning back
4. Recursion is often used for problems like factorial, Fibonacci, binary search, and tree traversals.

String

A string in C is a sequence of characters stored in a character array ending with a null character `'\0'`.

You can declare a string like this:

```
char name[20] = "Maria";
```

or input it from the user:

```
scanf("%s", name);
```

Example 1: Input and output of a string

```
#include<stdio.h>

int main()
{
    char name[50];
    printf("Enter your name: ");
    gets(name);   // gets() reads a full line (but unsafe, can use fgets
instead)
```

```
    printf("Your name is: %s", name);
    return 0;
}
```

Example 2: Find string length manually

```
#include<stdio.h>

int main()
{
    char str[100];
    int length = 0;
    printf("Enter a string: ");
    gets(str);

    for(int i = 0; str[i] != '\0'; i++)
        length++;

    printf("Length = %d", length);
    return 0;
}
```

Explanation:

The loop runs until the null character `'\0'` is found. Each step increases the length count.

Example 3: Reverse a string

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

int main()
{
    char str[100];
    printf("Enter a string: ");
    gets(str);

    int n = strlen(str);
    printf("Reversed string: ");
```

```
    for(int i = n - 1; i >= 0; i--)
        printf("%c", str[i]);

    return 0;
}
```

Example 4: Count vowels in a string

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

int main()
{
    char str[100];
    int count = 0;
    printf("Enter a string: ");
    gets(str);

    for(int i = 0; str[i] != '\0'; i++)
    {
        char ch = tolower(str[i]);
        if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u')
            count++;
    }

    printf("Total vowels = %d", count);
    return 0;
}
```

Key concepts of strings

A string is just an array of characters with `'\0'` marking the end.

You can use functions from `<string.h>` like:

`strlen(str)` – find length

`strcpy(dest, src)` – copy string

`strcat(s1, s2)` – join two strings

`strcmp(s1, s2)` – compare two strings

Pointer

A pointer is a variable that holds the *address* of another variable.

You can access or modify the value stored at that address using the *dereference operator* `*`.

Declaration example:

```
int *p;
```

Assigning address:

```
p = &x;
```

Accessing value:

`*p` gives the value stored at the address of `x`.

Example 1: Basic pointer usage

```
#include<stdio.h>

int main()
{
    int a = 10;
    int *p;
    p = &a;   // store address of a in pointer

    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Value stored in pointer p = %p\n", p);
    printf("Value pointed by p = %d\n", *p);
    return 0;
}
```

Explanation:

`p` stores the address of `a`.

`*p` accesses the value at that address (which is 10).

Example 2: Pointer with array

```
#include<stdio.h>

int main()
{
    int arr[5] = {10, 20, 30, 40, 50};
    int *p = arr;

    printf("Array elements using pointer:\n");
    for(int i = 0; i < 5; i++)
        printf("%d ", *(p + i));
    return 0;
}
```

Explanation:

When you assign `p = arr`, it points to the first element of the array.

Then `*(p + i)` gives each element of the array.

Example 3: Swapping numbers using pointers

```
#include<stdio.h>

void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main()
{
    int a = 5, b = 10;
    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swap: a = %d, b = %d", a, b);
    return 0;
}
```

Explanation:

Instead of sending the values, we send the addresses of `a` and `b`.

Inside the function, `*x` and `*y` directly modify the actual values stored at those addresses.

Example 4: Pointer and function recursion

```
#include<stdio.h>

void printArray(int *p, int n)
{
    if(n == 0)
        return;
    printf("%d ", *p);
    printArray(p + 1, n - 1);
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    return 0;
}
```

Important concepts of pointers

1. `&` gives the address of a variable.
2. `*` gives the value at that address.
3. Pointer arithmetic works only on arrays and valid memory blocks.
4. `int *p` means p is a pointer to an integer.
5. Pointers are powerful for dynamic memory allocation and function arguments.

Most important programs to practice

1. Factorial using recursion
2. Fibonacci using recursion
3. Sum of digits using recursion

4. Reverse a string manually
5. Count vowels and consonants
6. Swap two numbers using pointers
7. Print array elements using pointers
8. Find the largest element in an array using pointers
9. Calculate string length without `strlen()`
10. Print array recursively