PROGRAMMING FUNDAMENTALS

WEEK 12: POINTERS IN C

TABLE OF CONTENTS

RECURSIVE FUNCTIONS

Definition:

- A function that calls itself to solve a smaller version of the same problem.

Must have:

- Base case (to stop recursion)
- Recursive case (calls itself with simpler input)

RECURSIVE FUNCTIONS

Example:

```
int factorial(int n) {
    if (n == 0)                 // Base case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive call
}
```

```
int main() {
    printf("5! = %d\n", factorial(5));  // 120
    return 0;
}
```

WHAT IS A POINTER?

- **Core Idea:** A pointer is a variable that stores a memory address.

Why care?

- Modify variables in functions
- Build dynamic data structures
- Pass large data efficiently

WHAT IS A POINTER?

Syntax:

- ```
  int num = 25;
  int *ptr = #    // ptr holds address of num
  ```

Operators:

- & → "address of“
- * → "value at address" (dereference)

```
Variable:   num = 25      → stored at address 1000
Pointer:    ptr = 1000    → *ptr = 25
```
```

WHAT IS A POINTER?

Syntax:

`Num OR *ptr  num variable value`

`&num OR ptr  num variable address`

For Declaration: `int *ptr`

For usage of value: `*ptr`

For usage of address: `ptr`

POINTER TO DIFFERENT TYPES (INT/FLOAT/CHAR)

All pointers store addresses, but type matters for arithmetic and dereferencing.

TYPE	DECLARATION	EXAMPLE
Int	`Int *p;`	`Int a = 20;` `p = &a;`
Float	`Float *p;`	`Float b = 2.4;` `p = &b;`
char	`Char *p;`	`Char c = ‘A’;` `p = &c;`

POINTER TO DIFFERENT TYPES (INT/FLOAT/CHAR)

Live Demo:

```
int a = 5;
float b = 2.5;
char c = 'X’;

int *pi = &a;
float *pf = &b;
char *pc = &c;

printf("int: %d\n", *pi);    // 5
printf("float: %.1f\n", *pf); // 2.5
printf("char: %c\n", *pc);    // X
```

POINTERS TO ARRAYS

In C, array name = address of first element → it's a pointer!

Example:

```
int arr[3] = {10, 20, 30};
int *p = arr;  // same as: int *p = &arr[0];
```

Accessing Elements:

```
printf("%d\n", *p);       // 10 (arr[0])  □ *(p+i) □ p[i]/arr[i]
printf("%d\n", *(p+1));   // 20 (arr[1])
printf("%d\n", p[2]);     // 30 (yes! p[2] works)
```

POINTERS TO ARRAYS

Pointer Arithmetic:

- p + 1 → moves by sizeof(int) bytes (e.g., 4 bytes)
- So *(p + i) == arr[i]

Visual:

```
arr: [10][20][30]
      ↑
      p □ *(p+0)=10, *(p+1)=20, *(p+2)=30
```

POINTERS TO POINTERS

A pointer that stores the address of another pointer.

- Why? Needed for 2D arrays, dynamic allocation, advanced data structures.

Syntax:

```
int a = 100;
int *p = &a;        // p → holds address of a
int **pp = &p;      // pp → holds address of p
```

POINTERS TO POINTERS

Dereferencing:

```
printf("%d\n", a);      // 100
printf("%d\n", *p);     // 100
printf("%d\n", **pp);   // 100
```

Visual:

```
a = 100  ← at 1000
p = 1000 ← at 2000
pp = 2000
```

POINTERS TO POINTERS

🧠 Think:

- p = “remote to TV”
- pp = “remote to the remote”

⚠️ Don’t overuse — but understand it for future topics!

BUBBLE SORT USING POINTERS

Apply pointers to a real algorithm!

Goal: Sort an array by swapping elements using pointers.

Why pointers? Avoid array indexing; practice address manipulation.

BUBBLE SORT USING POINTERS

Code:

```
void bubbleSort(int *arr, int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (*(arr + j) > *(arr + j + 1)) {
                // Swap using pointers
                int temp = *(arr + j);
                *(arr + j) = *(arr + j + 1);
                *(arr + j + 1) = temp;
            }
        }
    }
}
```

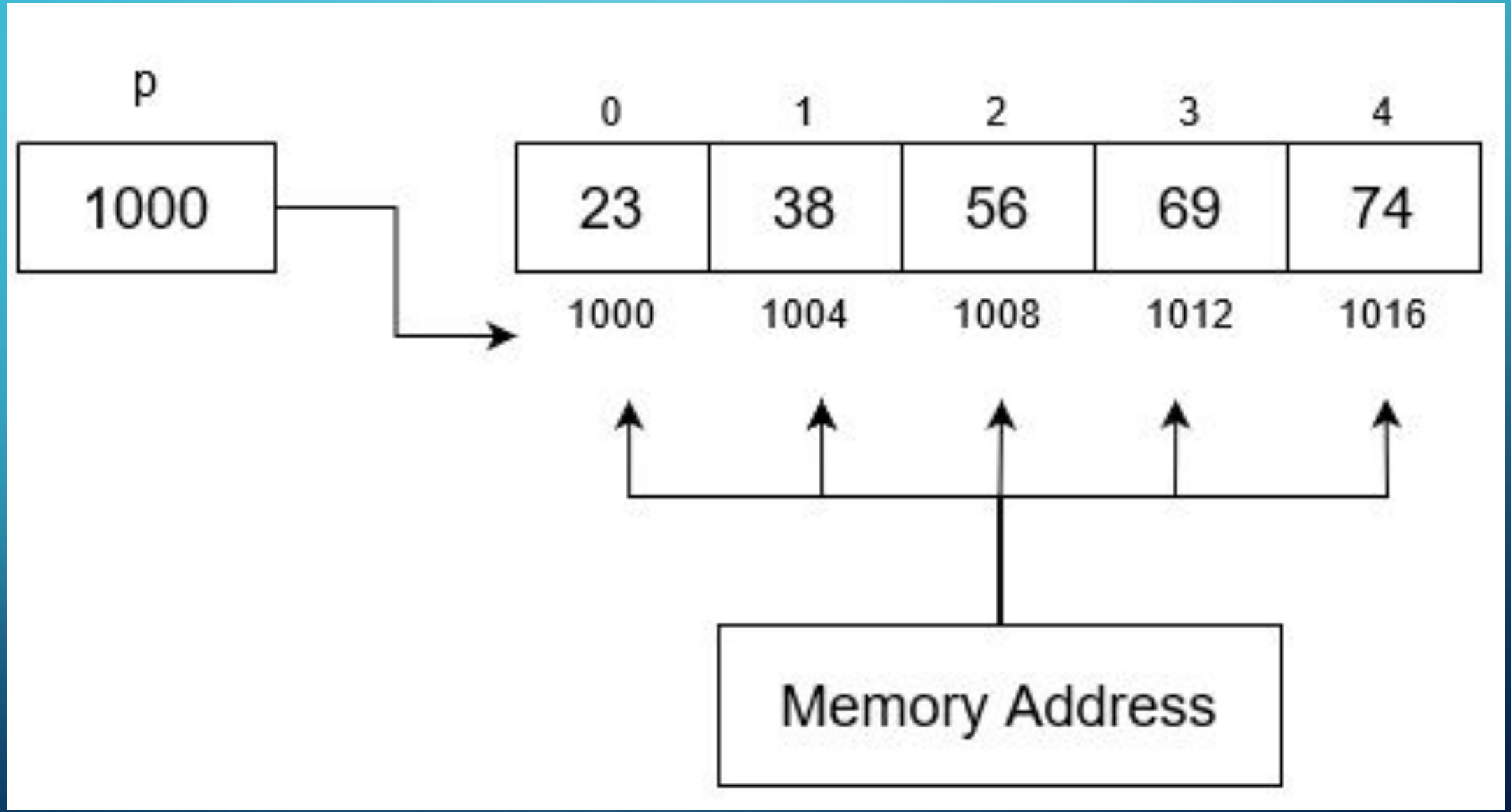
BUBBLE SORT USING POINTERS

In main():

```
int data[] = {64, 34, 25, 12, 22, 11, 90};
int n = sizeof(data)/sizeof(data[0]);
bubbleSort(data, n);
// Print sorted array
```

Note:

***(arr + j) is equivalent to arr[j]**


p
1000
0
1
2
3
4
23
38
56
69
74
1000
1004
1008
1012
1016
Memory Address

COMMON MISTAKES

Danger Zone!

Uninitialized pointer:

```
int *p; *p = 10;  // CRASH! (wild pointer)
```

Wrong pointer type:

```
int a = 5; float *p = &a;  // Wrong! Type mismatch.
```

Going out of bounds:

```
int arr[3]; int *p = arr; *(p + 10) = 99; // Undefined!
```

BEST PRACTICES

1. Always initialize pointers (to NULL or valid address).
2. Match pointer type with data type.
3. Use arr[i] for clarity; use *(arr+i) to understand memory.

ASSIGNMENT

1. Write a recursive function sumNatural(int n) that returns the sum of the first n natural numbers:
   `1 + 2 + 3 + ... + n` Include a proper base case. In main(), test it with n = 5 (expected output: 15).

2. Declare an integer, float, and char. Create pointers to each and print their values using dereferencing.

Instructions:
Submit as .c files

ASSIGNMENT

3. Given int arr[5] = {1,2,3,4,5};, use a pointer to print all elements without using [].

4. Write a function that takes a char* (pointer to char) and prints the character it points to.

5. (Challenge) Modify bubbleSort to accept a pointer and sort in descending order.

Instructions:
Submit as .c files