Handbook: Core Concepts in C Programming

1. Pointers

Definition:
A pointer is a variable that stores the memory address of another variable. It "points" to the location of data in memory.

Key Concepts:

- **Declaration**:

```
int *ptr; // Declares a pointer to an integer
```

- **Initialization**:

```
int num = 10;
ptr = &num; // ptr now holds the address of num
```

- **Dereferencing**:
  Access the value at the address using `*`.

```
printf("%d", *ptr); // Output: 10
```

- **Pointer Arithmetic**:
  Incrementing a pointer moves it to the next address (e.g., `ptr++` advances by `sizeof(int)` bytes).

Use Cases:

- Dynamic memory allocation (e.g., `malloc`).
- Efficient array/string manipulation.
- Passing large data to functions without copying.

Common Pitfalls:

- **Dangling Pointers**: Pointers pointing to deallocated memory.
- **Memory Leaks**: Forgetting to free dynamically allocated memory.
- **Null Pointers**: Always initialize pointers to `NULL` to avoid undefined behavior.

Best Practices:

- Use `free()` after `malloc/calloc` to prevent leaks.
- Check for `NULL` before dereferencing.

---

2. Memory Management

Memory Segments in C:

1. **Stack**: Stores local variables (auto-deallocated).
2. **Heap**: Dynamic memory (manually managed via `malloc`, `free`).
3. **Data Segment**: Global/static variables.
4. **Text Segment**: Code (read-only).

Heap Management Example:

```
int *arr = (int*)malloc(5 * sizeof(int)); // Allocate
if (arr == NULL) { /* Handle error */ }
free(arr); // Deallocate
```

Key Issues:

- **Stack Overflow**: Excessive recursion/local variables.
- **Heap Fragmentation**: Inefficient memory use over time.

---

3. Inbuilt C Libraries

Common Libraries & Functions:

1. **`stdio.h`**:
   - `printf()`, `scanf()`: Input/output.
   - `FILE*` functions: `fopen()`, `fclose()`.
2. **`stdlib.h`**:
   - `malloc()`, `free()`: Heap memory.
   - `atoi()`, `rand()`: Conversions/RNG.
3. **`string.h`**:
   - `strcpy()`, `strlen()`: String operations.
   - `memcpy()`: Copy memory blocks.
4. **`math.h`**:
   - `sqrt()`, `pow()`: Mathematical operations.
5. **`ctype.h`**:
   - `isalpha()`, `tolower()`: Character checks.

---

4. Call by Value vs. Call by Reference

Call by Value:

- Copies the argument's value into the function.
- Original variable remains unchanged.

```
void swap(int a, int b) { /* ... */ } // No effect outside
```

Call by Reference (Simulated via Pointers):

- Passes the address of the variable.
- Directly modifies the original data.

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

When to Use:

- **Value**: Small data (e.g., `int`), no side effects needed.
- **Reference**: Large data (e.g., structs), modify variables.

---

5. Recursion

Definition: A function that calls itself until a base case is reached.

Example: Factorial

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

Pros & Cons:

- ✅ Simplifies code for problems like tree traversal.
- ❌ Stack overflow risk; higher memory usage.

Best Practices:

- Always define a base case.

- Avoid deep recursion for large inputs.

---

6. Iteration

Definition: Repeating a block of code using loops (`for` , `while` , `do-while`).

Example: Factorial with Loop

```
int factorial(int n) {
  int result = 1;
  for (int i=1; i<=n; i++)
    result *= i;
  return result;
}
```

Recursion vs. Iteration:

Factor	Recursion	Iteration
Readability	Elegant for certain problems	Straightforward
Memory Efficiency	Uses stack (risk of overflow)	Minimal memory
Performance	Slower (function call overhead)	Faster

When to Use:

- **Iteration**: Performance-critical tasks, large data.
- **Recursion**: Problems with recursive structure (e.g., DFS).

Summary Cheat Sheet

- **Pointers**: Store addresses; use `*` and `&` .
- **Memory**: Manage heap with `malloc` / `free` ; avoid leaks.
- **Libraries**: Use `stdio.h` , `stdlib.h` for I/O and memory.
- **Call by Value/Reference**: Use pointers to modify variables.
- **Recursion**: Base case + recursive step; watch stack depth.
- **Iteration**: Faster, uses loops; prefer for simple tasks.