Functions & Pointers in C - Illustrated [Notes]

1. Function Basics

Functions are reusable blocks of code that make programs modular and readable.

Syntax: *return_type function_name(parameters) { body }*

Example:

```c
int add(int a, int b) {
    return a + b;
}
```

2. Parameter Passing

Call by Value: A copy of the argument is passed. **Call by Reference:** Using pointers, the function modifies the original variable.

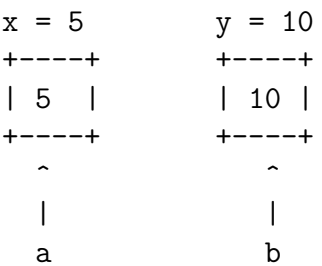
3. Pointers & Memory Addressing

- Declaration: `int *p;` - Initialization: `p = &x;` - Dereferencing: `*p` accesses or modifies the value at the stored address.

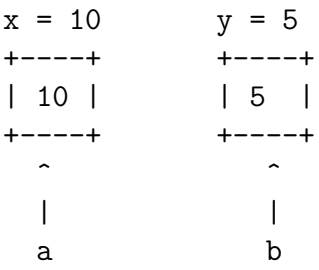
4. Practical Example: Swap Function

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    printf("%d %d", x, y); // Output: 10 5
}
```

Memory Layout (before swap):



Memory Layout (after swap):

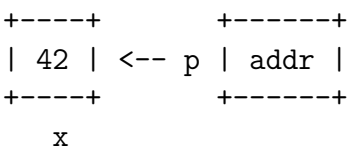


5. Example: Pointer Dereferencing

```c
int x = 42;
int *p = &x;

printf("%d", *p); // prints 42
```

Memory Diagram:

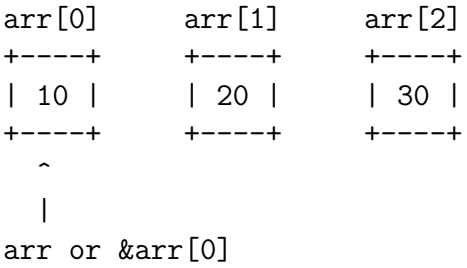


6. Arrays and Pointers

- An array name acts like a pointer to its first element. - Access: `arr[i]` is equivalent to `*(arr + i)`

```c
int arr[3] = {10, 20, 30};
printf("%d", *(arr + 1)); // prints 20
```

Memory Layout:



7. Dynamic Memory Allocation (Intro)

Pointers allow creation of memory at runtime.

```c
int *ptr;
ptr = (int*) malloc(3 * sizeof(int));
```

```
ptr[0] = 5;
ptr[1] = 15;
ptr[2] = 25;

free(ptr); // release memory
```

8. Linked List Basics

A linked list is a collection of nodes, where each node contains data and a pointer to the next node.

```
struct Node {
    int data;
    struct Node *next;
};

int main() {
    struct Node n1, n2, n3;

    n1.data = 10; n1.next = &n2;
    n2.data = 20; n2.next = &n3;
    n3.data = 30; n3.next = NULL;

    struct Node *head = &n1;
    while(head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
}
```

Memory Diagram:

```
+------+     +------+     +------+
| 10   | --> | 20   | --> | 30   | --> NULL
+------+     +------+     +------+
```

9. Common Pointer Errors

a) Uninitialized Pointer

```
int *p;      // uninitialized
*p = 5;      // ERROR: random memory location
```

b) Dangling Pointer

```
int *p = (int*) malloc(sizeof(int));
free(p);
*p = 10;     // ERROR: using freed memory
```

c) NULL Pointer Dereference

```
int *p = NULL;
printf("%d", *p); // ERROR: segmentation fault
```

d) Array Out of Bounds

```
int arr[3] = {1,2,3};
printf("%d", arr[5]); // ERROR: invalid memory access
```

e) Multiple Free

```
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);  // ERROR: double free
```

10. Why Pointers Matter

- Dynamic memory management - Efficient array and string handling - Building data structures (linked lists, trees) - Direct hardware/memory access

11. Best Practices

- Always initialize pointers before use - Avoid dangling pointers - Set to `NULL` after `free` - Check if `malloc` returned `NULL` - Free dynamically allocated memory

12. Summary Table

Concept	Description
Function Declaration	`return_type name(params);` defines a function
Call by Value	Copies arguments; original variables unchanged
Pointers & References	Use addresses to directly modify original variables
Swapping via Pointers	Example using `swap(int *a, int *b)`
Arrays and Pointers	`arr[i] == *(arr+i)`
Dynamic Memory Allocation	`malloc, calloc, realloc, free`
Linked List Basics	Nodes linked via `next` pointers
Pointer Errors	Uninitialized, dangling, NULL deref, OOB, double free
Pointer Safety	Initialize, check, free, nullify

13. TikZ Flowchart: Pointer Lifecycle