1. Pointers in C

Definition

A pointer is a variable that holds the memory address of another variable. Pointers provide a way to directly access and manipulate memory, which can lead to more efficient programs, especially when dealing with large data structures.

Referencing and Dereferencing

- **Referencing**: The address-of operator (`&`) is used to obtain the address of a variable.

  ```c
  int num = 42;
  int *ptr = &num; // ptr now holds the address of num
  ```

- **Dereferencing**: The dereference operator (`*`) is used to access the value at the address stored in a pointer.

  ```c
  int value = *ptr; // value is now 42
  *ptr = 100; // The value of num is now changed to 100
  ```

2. Extended Concepts of Pointers

- **Pointer Arithmetic**: You can perform arithmetic operations on pointers. When you increment a pointer, it moves to the next element of the type it points to.

  ```c
  int arr[] = {10, 20, 30};
  int *ptr = arr; // Points to the first element
  printf("%d\n", *ptr); // Output: 10
  ptr++; // Now points to the second element
  printf("%d\n", *ptr); // Output: 20
  ```

- **Pointers to Pointers**: You can have pointers that point to other pointers.

  ```c
  int **ptr_to_ptr = &ptr; // A pointer that points to ptr
  ```

3. Scale Factor

The scale factor refers to the size of the data type the pointer points to. When you perform pointer arithmetic, the increment doesn't move by 1 byte but rather by the size of the data type pointed to.

For example, if you have an `int` pointer and increment it, it will move forward by `sizeof(int)` bytes (typically 4 bytes on many systems).

4. Store and Fetch Value from Memory Using Pointer

You can directly access and modify memory locations using pointers:

```c
int num = 50;
int *ptr = &num;
*ptr = 75; // This changes num to 75
```

5. Memory Management in C

C provides mechanisms to manage memory through dynamic allocation.

Memory Layout in C

- **Stack**: Used for static memory allocation (e.g., local variables, function calls).
- **Heap**: Used for dynamic memory allocation (e.g., using functions like `malloc`, `calloc`).

Local and Global Variables

- **Local Variables**: Life cycle is tied to the function they're declared in; stored in the stack.
- **Global Variables**: Exist for the lifetime of the program; stored in the data segment.

6. Static and Dynamic Binding

- **Static Binding** (early binding): The type of a variable is known at compile-time. Example includes function calls in C.
- **Dynamic Binding** (late binding): The type is resolved at runtime, which is more common in languages like C++ with polymorphism.

7. Dynamic Memory Allocation

Dynamic memory allocation allows you to request memory at runtime using the heap rather than the stack.

Functions for Memory Management

1. **malloc**: Allocates a specified number of bytes and returns a pointer to the first byte. Uninitialized memory is returned.

   ```c
   int *arr = (int *)malloc(5 * sizeof(int)); // Allocates memory for 5 integers
   ```

2. **calloc**: Allocates memory for an array of elements, initializing the memory to zero.

   ```c
   int *arr = (int *)calloc(5, sizeof(int)); // Allocates and initializes memory for 5 integers
   ```

3. **realloc**: Changes the size of previously allocated memory block. The contents are preserved, but you must handle the potential move.

   ```c
   arr = (int *)realloc(arr, 10 * sizeof(int)); // Resizes the array to hold 10 integers
   ```

4. **free**: Frees previously allocated memory to prevent memory leaks.

   ```c
   free(arr); // Frees the memory allocated to arr
   ```

Memory Management Example

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
```

```c
    int *arr;
    arr = (int *)malloc(5 * sizeof(int)); // Dynamically allocate an array of 5 integers

    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1; // Exit if allocation failed
    }

    // Store values in the array
    for (int i = 0; i < 5; i++) {
        arr[i] = i + 1; // Assign values 1 to 5
    }

    // Fetch values
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]); // Prints: 1 2 3 4 5
    }
    printf("\n");

    // Resize the array
    arr = (int *)realloc(arr, 10 * sizeof(int)); // Resize to hold 10 integers

    // Check if realloc succeeded
    if (arr == NULL) {
        printf("Memory reallocation failed!\n");
        return 1;
    }

    // Free the allocated memory
    free(arr);
    return 0;
}
```

Summary

1. **Pointers**: Store memory addresses and support operations like referencing and dereferencing.
2. **Memory Management**: Essential for dynamic allocation using `malloc`, `calloc`, `realloc`, and `free`.
3. **Memory Layout**: Distinguishes between stack and heap memory, local and global variables.
4. **Static vs. Dynamic Binding**: Defines compile-time vs. runtime type resolution.