**Lab: Dynamic Memory Management in C**

**Objective:**

This lab will focus on various memory management techniques in C. You will work with:

* Dynamically adjusting memory with realloc()
* Understanding pointer concepts like pointer to constant and constant pointer
* Using void pointers and understanding alignment restrictions
* Working with generic pointer arguments in functions like memset() and memcpy()

**Pre-requisites:**

* Basic understanding of pointers in C
* Familiarity with malloc(), free(), and basic memory management concepts

**1. Dynamically Adjusting Memory Using realloc()**

**Task:**

Write a program to dynamically allocate memory for an integer array and adjust its size using realloc() as needed.

**Steps:**

1. Allocate memory for an array of 5 integers using malloc().
2. Assign values to the array and print them.
3. Use realloc() to resize the array to hold 10 integers.
4. Print the resized array, ensuring new elements are initialized correctly.
5. Free the allocated memory.

c

#include <stdio.h>

#include <stdlib.h>

int main() {

// Initial allocation

int \*arr = (int\*)malloc(5 \* sizeof(int));

if (arr == NULL) {

printf("Memory allocation failed\n");

return 1;

}

// Assign values

for (int i = 0; i < 5; i++) {

arr[i] = i + 1;

}

// Print initial values

printf("Initial Array: ");

for (int i = 0; i < 5; i++) {

printf("%d ", arr[i]);

}

printf("\n");

// Reallocate memory

arr = (int\*)realloc(arr, 10 \* sizeof(int));

if (arr == NULL) {

printf("Reallocation failed\n");

return 1;

}

// Assign new values

for (int i = 5; i < 10; i++) {

arr[i] = i + 1;

}

// Print resized array

printf("Resized Array: ");

for (int i = 0; i < 10; i++) {

printf("%d ", arr[i]);

}

printf("\n");

// Free memory

free(arr);

return 0;

}

**2. Pointer to a Constant vs. Constant Pointer**

**Task:**

Understand the difference between a **pointer to a constant** and a **constant pointer**. Write a program to demonstrate both.

**Steps:**

1. A pointer to a constant allows modifying the pointer, but not the data it points to.
2. A constant pointer allows modifying the data, but not the pointer itself.

c

#include <stdio.h>

int main() {

int a = 10, b = 20;

// Pointer to a constant

const int \*ptr1 = &a;

// ptr1 = &b; // Allowed

// \*ptr1 = 30; // Error: Cannot modify the value of 'a'

// Constant pointer

int \*const ptr2 = &a;

// ptr2 = &b; // Error: Cannot change the address stored in 'ptr2'

\*ptr2 = 30; // Allowed: Modifies the value of 'a'

printf("Value of a: %d\n", a);

printf("Value of b: %d\n", b);

return 0;

}

**3. Void Pointers and Alignment Restrictions**

**Task:**

Use a void pointer to handle different data types, and understand potential alignment restrictions.

**Steps:**

1. Declare a void pointer and assign it to different data types (e.g., int, double).
2. Demonstrate dereferencing the void pointer after casting to the correct type.
3. Discuss alignment issues (e.g., why double might need to be aligned on a 8-byte boundary).

c

#include <stdio.h>

int main() {

int a = 10;

double b = 3.14;

// Void pointer

void \*ptr;

// Pointing to an integer

ptr = &a;

printf("Value of a (int): %d\n", \*(int\*)ptr);

// Pointing to a double

ptr = &b;

printf("Value of b (double): %.2f\n", \*(double\*)ptr);

return 0;

}

**4. Generic Pointer Argument in memset() and memcpy()**

**Task:**

Demonstrate the use of memset() and memcpy() with generic pointers.

**Steps:**

1. Use memset() to set the values of an array to a specific byte (e.g., setting an array of integers to 0).
2. Use memcpy() to copy data between two arrays of different types (e.g., copying an array of integers into an array of char).

c

#include <stdio.h>

#include <string.h>

int main() {

// Using memset() to set an array to 0

int arr[5];

memset(arr, 0, 5 \* sizeof(int)); // Set all bytes to 0

printf("Array after memset: ");

for (int i = 0; i < 5; i++) {

printf("%d ", arr[i]);

}

printf("\n");

// Using memcpy() to copy data between arrays

char src[] = "Hello";

char dest[6]; // Must be large enough to hold the string

memcpy(dest, src, 6); // Copying 6 bytes, including null terminator

printf("Destination string after memcpy: %s\n", dest);

return 0;

}

**Lab Questions:**

1. What happens when you try to resize the memory block using realloc() to a size smaller than the original allocation?
2. How does a pointer to a constant differ from a constant pointer? Explain with an example.
3. What is the significance of alignment when using void pointers? Provide an example where misalignment could cause issues.
4. How does memset() work, and what happens when you pass a void pointer to it?
5. How does memcpy() handle different data types when copying between arrays? What considerations should be made when using memcpy() for different types?