| CL1002<br>*Programming Fundamentals Lab* | Lab 12<br>Introduction to Pointers and Dynamic Memory Allocation |
|---|---|

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2024

Author: Muhammad Nouman Hanif

## AIMS AND OBJECTIVES

To understand the concepts of pointers and dynamic memory allocation in C and apply them to create, manipulate, and free memory dynamically during runtime.

## INTRODUCTION

**Pointers and dynamic memory allocation** are fundamental concepts in C programming that provide programmers with powerful tools for **efficient memory management and flexibility**. Unlike static memory allocation, where the size of memory is determined at **compile-time**, dynamic memory allocation allows the program to request and manage memory at **runtime based** on actual requirements.

**Pointers**, which store the **memory address of another variable**, are at the core of dynamic memory allocation. By leveraging pointers, developers can create data structures of varying sizes, manage large datasets, and optimize the usage of system memory. However, with great power comes responsibility: improper use of pointers and dynamic memory can lead to errors such as **memory leaks, segmentation faults, and undefined behavior**.

## POINTER DECLARATION

With the theory in place let's look at some examples.

Pointer Declaration:

```
01 #include <stdio.h>
02
03 int main() {
04     int a = 10;
05     int *p;
06     p = &a;
07     printf("Value of a: %d\n", a);
08     printf("Address of a: %d\n", &a);
09     printf("Value of p (Address stored in pointer):
                  %d\n", p);
10     printf("Address of a: %p\n", &a); // Use %p
11     printf("Value of p: %p\n", p);

12     printf("Value pointed to by p: %d\n", *p);
13
14     return 0;
15 }
```

**Note:**

**&** (Address-of operator): Gets the memory address of a variable.

Example: p = **&**a; assigns the address of a to p.

***** (Dereference operator): Accesses the value at the memory address stored in the pointer.

Example: *p gives the value of a in this case.

**Pointers and Arrays:**

Pointers and arrays are closely related. Here's an example to illustrate:

```
01 #include <stdio.h>
02
03 int main() {
04     int arr[] = {10, 20, 30};
05     int *p = arr;
06     printf("Address of arr[0]: %d, Value: %d\n\n",
           p, arr);
07     printf("Address of arr[0]: %d, Value: %d\n", p,
           *arr);

08     printf("Address of arr[0]: %d, Value: %d\n\n",
09     p, *(arr+1));

10     printf("Address of arr[0]: %d, Value:%d\n",p, *p);
11     printf("Address of arr[1]: %d, Value:%d\n",
                   (p + 1), *(p + 1));
12     printf("Address of arr[2]: %d, Value: %d\n",
                   (p + 2), *(p + 2));
13
14     return 0;
15 }

16
```

**Note:**

When declaring an array to a variable it's a reference to memory which is the first block of the **contagious memory**. Adding/Subtracting into this reference changes the **starting point** of the array.

**Null Pointers:**

Pointers which **point to nothing**. They exist as placeholders for later use.

```
01 #include <stdio.h>
02
03 int main() {
04     int *p = NULL;    // Null pointer, points to nothing
05
06     if (p == NULL) {
07         printf("Pointer is NULL, it does not point to any
                   valid memory location.\n");
08     }
09
10     return 0;
11 }
```

**Pass by Value and Pass by Reference:**

**1. Understanding the Concepts**

**Pass by Value:**

- When a function is called with arguments passed by value, a **copy of the actual data** is made.
- Any changes made to the parameter inside the **function do not affect** the original data outside the function.
- Only the copy in the local scope of the function is modified.

**Key Characteristics:**

- **Original Value Unchanged**: The original variable remains unaffected.
- **Memory Usage**: Requires extra memory for the copy.

```
01 void modifyValue(int x) {
02     x = 10;  // Only modifies the local copy
03 }
04 int main() {
05     int num = 5;
06     modifyValue(num);
07     printf("%d", num);  // Output: 5
08     return 0;
09 }
```

**Pass by Reference:**

- When a function is called with arguments passed by reference, the function receives a reference (or address) to the actual data.
- Changes made to the parameter inside the function **directly** affect the original data outside the function.

**Key Characteristics:**

- **Original Value Modified**: The original variable is directly manipulated.
- **Efficient Memory Usage**: No duplicate copy is created, only the reference (pointer) is passed.

```
10 void modifyValue(int *x) {
11     *x = 10;  // Modifies the actual variable through the
   pointer
12 }
13 int main() {
14     int num = 5;
15     modifyValue(&num);
16     printf("%d", num);  // Output: 10
17     return 0;
18 }
```

**2. Using Pointers for Pass by Reference**:

- By passing a pointer to a function, we allow the function to modify the variable's value directly.
- The & operator is used to pass the address of a variable.

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main() {
    int x = 5, y = 10;
    swap(&x, &y);  // Pass addresses
    printf("x = %d, y = %d", x, y);  // Output: x =
       10, y = 5
    return 0;
}
```

**3. When to Use Which?**

- **Pass by Value**:
  - Use when the function does not need to modify the original data.
  - Safe from unintended side effects.
- **Pass by Reference**:
  - Use when the function needs to modify the original data.
  - Efficient for large data structures like arrays or objects.

```
#include <stdio.h>
struct Point {
    int x;
    int y;
};
int main() {
    struct Point p1 = {10, 20};  // Initialize a
      structure variable
    struct Point *ptr = &p1; // Pointer to the
        structure
    // Access and modify structure members using the
       pointer
    printf("Original Point: x = %d, y = %d\n", ptr->x,
    ptr->y);
        ptr->x = 30;  // Modify x through pointer
    ptr->y = 40;  // Modify y through pointer
    printf("Modified Point: x = %d, y = %d\n", ptr->x,
        ptr->y);
    return 0;
}
```

## 2D POINTERS

Previously, we discussed how arrays are essentially pointers to memory blocks, but what about 2-dimensional pointers or an array of pointers.

**Concept 1: Pointer to a 2D Array**

A 2D array is essentially an array of arrays stored in contiguous memory. A pointer can be used to **traverse and manipulate this memory**.

```
01 #include <stdio.h>
02
03 int main() {
04     int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };
05     int (*p)[3];
06
07     p = arr;
08
09     printf("Accessing elements using the pointer:\n");
10     printf("arr[0][0]: %d, arr[0][1]: %d, arr[0][2]:
                    %d\n", (*p)[0], (*p)[1], (*p)[2]);
11     printf("arr[1][0]: %d, arr[1][1]: %d, arr[1][2]:
           %d\n", (*(p + 1))[0],(*(p + 1))[1],(*(p +
   1))[2]);
12
13     return 0;
14 }
```

**Explanation**

1. **int (*p)[3];**: Declares a pointer to an array of 3 integers.
2. **p = arr;**: Assigns the base address of the first row of the 2D array to p.
3. Use *p to access rows and (*p)[col] to access individual elements.

**Concept 2: Array of Pointers**

In some cases, you may want to store a list of arrays, each of **varying lengths**. This is achieved using an **array of pointers**.

```
01 #include <stdio.h>
02
03 int main() {
04     int row1[] = {1, 2, 3};
05     int row2[] = {4, 5};
06     int row3[] = {6, 7, 8, 9};
07
08     int *arr[] = {row1, row2, row3}; // Array of pointers
09
10     printf("Accessing elements from the array of
                pointers:\n");
11     printf("arr[0][2]: %d\n", arr[0][2]);
12     printf("arr[1][1]: %d\n", arr[1][1]);
13     printf("arr[2][3]: %d\n", arr[2][3]);
14
15     return 0;
16 }
```

**Explanation**

1. **int *arr[] = {row1, row2, row3};**: Declares an array of pointers where each pointer points to a row.
2. Use arr[row][col] syntax for direct access to elements.

## VOID POINTERS

A **void pointer** is a special type of pointer that can point to **any data type**. This is in contrast to normal data type pointers, which are bound to point to **specific data** types like int, float, or char.

Declaration of Void Pointers:

```
01 void *ptr;
```

**Note:**

- **Generic Pointers**: These pointers are called Generic Points and can hold the address of any data type.
- **Typecasting Required**: Since the type is not known, void pointers **cannot be dereferenced directly;** you must **typecast** them to the appropriate type first.
- **Memory Management**: Often used in dynamic memory allocation functions like malloc() and free().

## EXAMPLE

```
01 #include <stdio.h>
02
03 int main() {
04     int a = 10;
05     float b = 5.5;
06     char c = 'X';
07
08     void *ptr; // Declare a void pointer
09
10     // Point to different data types
11     ptr = &a;
12     printf("Value of a using void pointer: %d\n",
                    *(int *)ptr);
13
14     ptr = &b;
15     printf("Value of b using void pointer: %.2f\n",
                   *(float *)ptr);
16
17     ptr = &c;
18     printf("Value of c using void pointer: %c\n",
                  *(char *)ptr);
19
20     return 0;
21 }
```

**When to Use Void Pointers?**

**Generic Functions**: Functions like malloc() and free() **return void *** because they don't depend on the type of data being allocated or freed.

```
01 int *p = (int*) malloc(5 * sizeof(int));
```

**Generic Data Structures**: For example, when implementing linked lists, stacks, or queues that can store any type of data.

## EXAMPLE

```
01 #include <stdio.h>
02
03 void printValue(void *ptr, char type) {
04     switch (type) {
05         case 'i': printf("Integer: %d\n", *(int
                          *)ptr); break;
06         case 'f': printf("Float: %.2f\n", *(float
                                 *)ptr); break;
07         case 'c': printf("Character: %c\n", *(char
                                *)ptr); break;
08         default: printf("Unknown type\n");
09     }
10 }
11
12 int main() {
13     int a = 42;
14     float b = 3.14;
15     char c = 'A';
16
17     printValue(&a, 'i');
18     printValue(&b, 'f');
19     printValue(&c, 'c');
20
21     return 0;
22 }
```

**Typecasting:**

Typecasting is the process of converting a variable from **one data type to another** in C. It is used when a program requires data to be interpreted or operated upon as a different type than its original.

**Types of Typecasting:**

**1. Implicit Typecasting (Type Promotion)**

- Also known as **type coercion**, this is automatically performed by the compiler.
- It usually occurs when data of a smaller data type is assigned to a larger data type.

```
01    int a = 10;
02 float b = a; // 'a' is implicitly converted to float
```

- No data loss occurs in implicit typecasting.

**2. Explicit Typecasting**

- The programmer manually specifies the conversion type.
- Syntax: (type) expression
- Example:

```
01    float a = 5.75;
02 int b = (int)a; // Explicitly cast 'a' to int
```

- Data loss may occur, depending on the types involved.

```
01 #include <stdio.h>
02 int main()
03 {
04     // Given a & b
05     int a = 15, b = 2;
06     char x = 'a';
07     double div;
08     // Explicit Typecasting in double
09     div = (double)a / b;
10     // converting x implicitly to a+3 i.e, a+3 = d
11     x = x + 3;
12
13     printf("The result of Implicit typecasting is
14     %c\n", x);
15     printf("The result of Explicit typecasting is %f",
16     div);
17     return 0;
18 }
```

## DYNAMIC MEMORY ALLOCATION (DMA)

Dynamic Memory Allocation (DMA) allows you to **allocate memory during runtime**, providing flexibility to create data structures like **arrays or linked lists** when the **size is not known at compile time**. This is particularly useful when dealing with **varying data sizes** or optimizing memory usage.

There are two types of memory allocation in C.

**Compile-Time Memory Allocation (Static Allocation):**

- Memory is allocated for variables and data structures at compile time.
- The size is fixed and **cannot change during execution**.
- Done in:
  - **Global Memory**: For global and static variables.
  - **Stack Memory**: For local variables and function call-related data.

**Example**:

```
01 int a = 10;  // Compile-time allocation
02 int arr[5];  // Array of fixed size
```

**Run-Time Memory Allocation (Dynamic Allocation):**

- Memory is allocated during execution using functions like **malloc, calloc, and realloc**.
- The size and lifetime of the memory are determined at runtime, offering flexibility.
- Managed in the **Heap Memory**.

**Example:**

```
01 int *ptr = (int *)malloc(sizeof(int) * 5);  // Runtime
   allocation
```

Side-by-Side comparison

| Aspect | Compile-Time Allocation | Dynamic Memory Allocation |
|---|---|---|
| When allocated | At compile time. | At runtime. |
| Memory Source | Stack or global memory. | Heap memory. |
| Size Flexibility | Fixed size (must be known at compile time). | Can change during execution using realloc. |
| Management | Automatically managed by the compiler. | Must be explicitly managed by the programmer. |
| Lifetime | Limited to scope for local variables or static for globals. | Persists until explicitly freed using free. |
| Examples | int a[10];, int x = 20; | int *p = (int *)malloc(10 * sizeof(int)); |

**Static vs Dynamic Allocation in a Program**

Compile-Time (Static Allocation) Example:

```c
#include <stdio.h>

int main() {
    int arr[5];  // Fixed array, size cannot change

    for (int i = 0; i < 5; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }

    // Cannot add more elements dynamically
    return 0;
}
```

Dynamic Memory Allocation Example:

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

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;  // Initialize elements
        printf("%d ", arr[i]);
    }

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

**Note:** In the static example the **size of the array is fixed**, while in the dynamic example the **size of the array is determined at runtime**.

**Question to you:**
What if I try to allocate the memory like this

```c
int n;
scanf("%d",&n);
int arr[n]
```

Is this a compile time memory allocation or runtime memory allocation?

**Methods to allocate memory in C:**

There exists three primary ways to allocate memory in runtime in C.

**malloc (Memory Allocation)**

- Allocates a **single block of memory** of the specified size (in bytes).
- Memory is **not initialized** (contains garbage values).
- Returns a pointer to the allocated memory or **NULL** if allocation fails.

```
01 int *ptr = (int *)malloc(5 * sizeof(int)); // Allocates
   memory to already garbage value for 5 integers
02 if (ptr == NULL) {
03     printf("Memory allocation failed!\n");
04 }
```

**calloc (Contiguous Allocation)**

- Allocates memory for an **array of elements**, each of a specified size.
- Memory is **initialized to zero**.
- Useful for initializing **arrays or buffers**.

```
01 int *ptr = (int *)calloc(5, sizeof(int)); // Allocates
   and initializes memory to Zero for 5 integers
02 if (ptr == NULL) {
03     printf("Memory allocation failed!\n");
04 }
```

**realloc (Reallocation)**

- Resizes a previously allocated memory block (using malloc or calloc).
- Can **increase or decrease the size** of the memory block.
- If the new size is larger:
    - Data in the original block is preserved.
    - The new portion remains uninitialized.
- If reallocation fails, it returns NULL without affecting the original block.

```
01 int *ptr = (int *)malloc(5 * sizeof(int));
02 ptr=(int*)realloc(ptr,10*sizeof(int)); //Resize to 10
03 if (ptr == NULL) {
04     printf("Reallocation failed!\n");
05 }
```

## MALLOC EXAMPLE

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

int main() {
    int *ptr;
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory using malloc
    ptr = (int *)malloc(n * sizeof(int));

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

    // Input elements
    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &ptr[i]);
    }

    // Print elements
    printf("You entered:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }

    // Free the allocated memory
    free(ptr);

    return 0;
}
```

## CALLOC EXAMPLE

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

int main() {
    int *ptr;
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory using calloc
    ptr = (int *)calloc(n, sizeof(int));

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

    printf("Memory initialized to zero:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr);

    return 0;
}
```

## REALLOC EXAMPLE

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

int main() {
    int *ptr;
    int n1, n2, i;

    printf("Enter the initial number of elements: ");
    scanf("%d", &n1);

    ptr = (int *)malloc(n1 * sizeof(int));

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

    printf("Enter %d integers:\n", n1);
    for (i = 0; i < n1; i++) {
        scanf("%d", &ptr[i]);
    }

    printf("Enter the new size of the array: ");
    scanf("%d", &n2);

    // Resize the memory block using realloc
    ptr = (int *)realloc(ptr, n2 * sizeof(int));

    if (ptr == NULL) {
        printf("Memory reallocation failed!\n");
        return 1;
    }

    if (n2 > n1) {
        printf("Enter %d more integers:\n", n2 - n1);
        for (i = n1; i < n2; i++) {
            scanf("%d", &ptr[i]);
        }
    }

    printf("Updated array:\n");
    for (i = 0; i < n2; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr);

    return 0;
}
```

**Note:** The **free()** function is to **deallocate memory** when its functionality is no longer required to avoid memory leaks. Notice in the above-mentioned example There exists **dangling pointers** (*when a pointer continues to reference a memory location after it has been freed or is no longer valid*) and this should be set to NULL otherwise undefined behavior can occur.

**Example:**
Consider this code

```
01 int *ptr = (int *)malloc(sizeof(int));
02 *ptr = 10;
03 free(ptr);  // Memory is freed, ptr is now dangling
04 *ptr=20; // Undefined behavior: accessing freed memory
```

Because the pointer being free of the allocated memory cannot act as a traditional reference to a variable, we would need to set the pointer to NULL so we can use it as intended.

# Problems

**Exercise 1: Pointer and Array Manipulation**

Write a function that takes a pointer to an array of integers and the size of the array. The function should double every element of the array and return the modified array.

```
void doubleArray(int *arr, int size);
```

- In the main function, dynamically allocate an array using malloc(), and pass the pointer to the doubleArray function.
- Print the modified array before freeing the dynamically allocated memory.

**Exercise 2: Pointer to a 2D Array**

Create a 2D array using dynamic memory allocation with malloc(). The array should have 3 rows and 4 columns. Write a function that takes a pointer to a dynamically allocated 2D array and modifies its elements.

- The function should double every element of the array.
- After calling the function, print the modified array and free the allocated memory.

**Exercise 3: Passing Structures via Pointer**

Define a structure struct Rectangle that has two members: width and height (both integers).

- Write a function that takes a pointer to a Rectangle structure and modifies its width and height.
- In main(), create an instance of Rectangle, pass its pointer to the function, and print the modified values of width and height.

**Exercise 4: Implementing Dynamic Arrays**

Write a program that allows the user to create a dynamic array of integers using malloc(). The program should allow the user to enter the size of the array and then populate the array with integers.

- After the user enters the values, print the array elements.
- Reallocate the array using realloc() to accommodate an additional 5 elements, then print the updated array.
- Free the dynamically allocated memory once done.

**Exercise 5: Void Pointer with Multiple Data Types**

Write a function that accepts a void pointer and a character indicating the type of data pointed to. Based on the type, print the value stored at the pointer.

- Use a switch statement to check the type and cast the pointer to the appropriate type.
- In the main() function, demonstrate this function by passing an integer, float, and character as arguments, and print each value correctly.

**Exercise 6: Use of calloc() and realloc()**

Write a program to dynamically allocate an array of n integers using calloc(). Initialize all elements to zero and print the values. Reallocate the array to double its size using realloc() and populate the new elements with values. Print the array before and after reallocation.

**Exercise 7: Pointer Arithmetic with Structures**

Define a structure struct Student with three members: name (string), age (int), and score (float). Create an array of 5 students and use pointer arithmetic to print the name and score of each student.

- Avoid using array indexing.
- Modify the program to input student details and display them using pointer arithmetic.

**Exercise 8: Handling Dynamic Memory Allocation Errors**

Write a function allocateMemory that uses malloc() to dynamically allocate memory for an array of integers. If malloc() fails, print an error message and return NULL. Otherwise, initialize the array with values and return the pointer.

- In the main() function, check if the allocation was successful. If it was, free the memory after use.