CL1002 ***Programming Fundamentals Lab***	Lab 12 Introduction to Pointers and Dynamic Memory Allocation

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2024

AIMS AND OBJECTIVES

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

INTRODUCTION

Pointers and dynamic memory allocation are fundamental concepts in C and 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 lets look at some examples.

Pointer Declaration:

```
#include <stdio.h>

int main() {
    int a = 10;
    int *p;
    p = &a;
    printf("Value of a: %d\n", a);
    printf("Address of a: %d\n", &a);
    printf("Value of p (Address stored in pointer):
              %d\n", p);
    printf("Value pointed to by p: %d\n", *p);

    return 0;
}
```

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:

```
#include <stdio.h>

   int main() {
       int arr[] = {10, 20, 30};
       int *p = arr;
       printf("Address of arr[0]: %d, Value:%d\n",p, *p);
       printf("Address of arr[1]: %d, Value:%d\n",
               (p + 1), *(p + 1));
       printf("Address of arr[2]: %d, Value: %d\n",
               (p + 2), *(p + 2));

       return 0;
   }

```

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.

```
#include <stdio.h>

int main() {
    int *p = NULL;     // Null pointer, points to nothing

    if (p == NULL) {
        printf("Pointer is NULL, it does not point to any
                  valid memory location.\n");
    }

    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.

```
#include <stdio.h>

   int main() {
       int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };
       int (*p)[3];

       p = arr;

       printf("Accessing elements using the pointer:\n");
       printf("arr[0][0]: %d, arr[0][1]: %d, arr[0][2]:
                 %d\n", (*p)[0], (*p)[1], (*p)[2]);
       printf("arr[1][0]: %d, arr[1][1]: %d, arr[1][2]:
      %d\n", (*(p + 1))[0],(*(p + 1))[1],(*(p + 1))[2]);

       return 0;
   }
```

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.

```c
#include <stdio.h>

int main() {
    int row1[] = {1, 2, 3};
    int row2[] = {4, 5};
    int row3[] = {6, 7, 8, 9};

    int *arr[] = {row1, row2, row3}; // Array of pointers

    printf("Accessing elements from the array of
           pointers:\n");
    printf("arr[0][2]: %d\n", arr[0][2]);
    printf("arr[1][1]: %d\n", arr[1][1]);
    printf("arr[2][3]: %d\n", arr[2][3]);

    return 0;
}
```

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:

```
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

```
#include <stdio.h>

   int main() {
       int a = 10;
       float b = 5.5;
       char c = 'X';

       void *ptr; // Declare a void pointer

       // Point to different data types
       ptr = &a;
       printf("Value of a using void pointer: %d\n",
               *(int *)ptr);

       ptr = &b;
       printf("Value of b using void pointer: %.2f\n",
              *(float *)ptr);

       ptr = &c;
       printf("Value of c using void pointer: %c\n",
             *(char *)ptr);

       return 0;
   }
```

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.

```c
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

```c
#include <stdio.h>

   void printValue(void *ptr, char type) {
       switch (type) {
           case 'i': printf("Integer: %d\n", *(int
                     *)ptr); break;
           case 'f': printf("Float: %.2f\n", *(float
                             *)ptr); break;
           case 'c': printf("Character: %c\n", *(char
                            *)ptr); break;
           default: printf("Unknown type\n");
       }
   }

   int main() {
       int a = 42;
       float b = 3.14;
       char c = 'A';

       printValue(&a, 'i');
       printValue(&b, 'f');
       printValue(&c, 'c');

       return 0;
   }
```

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:

```
int a = 10;  // Compile-time allocation
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:

```
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.

```
int *ptr = (int *)malloc(5 * sizeof(int)); // Allocates
memory for 5 integers
if (ptr == NULL) {
    printf("Memory allocation failed!\n");
}
```

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.

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

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.

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

MALLOC EXAMPLE

```
#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 and this should be set to NULL otherwise undefined behavior can occur.

Example:

Consider this code

```
int *ptr = (int *)malloc(sizeof(int));
   *ptr = 10;
   free(ptr);  // Memory is freed, ptr is now dangling
   *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