Module 4 : Pointers & Dynamic Array

By the end of this module, students will be able to:

- Understand the concept and purpose of pointers in C
- Declare and initialize pointers correctly
- Use pointer operators (& and *) effectively
- Perform pointer arithmetic operations
- Work with pointers and arrays
- Pass pointers to functions
- Understand the relationship between pointers and strings
- Allocate and manage dynamic memory
- Work with dynamic arrays
- Avoid common pointer-related errors and pitfalls
- Apply pointers in practical programming scenarios

- 1. Introduction to Pointers
- 2. Pointer Basics
- 3. Pointer Arithmetics
- 4. Pointers and Arrays
- 5. Pointers and Functions
- 6. Pointers and Strings
- 7. Dynamic Memory Allocation & Array
- 8. Common Pointer Pitfalls and Best Practices
- 9. Practical Examples with Dynamic Memory

1. Introduction to Pointers

1.1 What are Pointers?

A **pointer** is a variable that stores the memory address of another variable. Instead of holding a data value directly, a pointer "points to" the location in memory where the data is stored.

Analogy: Think of computer memory like a street with houses:

- Each house (memory location) has an address (memory address)
- Each house contains something (data value)
- A pointer is like writing down a house address on paper
- You can use that address to find and access the house

Why Use Pointers?

1. **Dynamic Memory Allocation:** Create variables at runtime
2. **Efficient Array/String Manipulation:** Access elements without copying
3. **Function Parameter Passing:** Modify variables from within functions
4. **Data Structures:** Build linked lists, trees, graphs, etc.
5. **System Programming:** Direct memory access and hardware interaction

1.2 Memory Addresses

Every variable in C is stored at a specific memory location, identified by a unique address.

```
#include <stdio.h>

int main() {
    int age = 25;
    float height = 5.9f;
    char grade = 'A';

    printf("Value of age: %d\n", age);
    printf("Address of age: %p\n", (void*)&age);

    printf("Value of height: %.1f\n", height);
    printf("Address of height: %p\n", (void*)&height);

    printf("Value of grade: %c\n", grade);
```

```
    printf("Address of grade: %p\n", (void*)&grade);

    return 0;
}

/* Output (addresses will vary):
Value of age: 25
Address of age: 0x7ffd5c8e4a3c
Value of height: 5.9
Address of height: 0x7ffd5c8e4a38
Value of grade: A
Address of grade: 0x7ffd5c8e4a37
*/
```

Key Points:

- Memory addresses are typically displayed in hexadecimal (base 16)
- The `&` operator gets the address of a variable
- Format specifier `%p` prints pointer/address values
- Adjacent variables may have addresses close to each other

2. Pointer Basics

2.1 Declaring Pointers

Syntax:

```
data_type *pointer_name;
```

Examples:

```
int *ptr;        // Pointer to an integer
float *fptr;     // Pointer to a float
char *cptr;      // Pointer to a character
double *dptr;    // Pointer to a double
```

Important Notes:

- The `*` (asterisk) indicates that the variable is a pointer
- The asterisk can be placed next to the type or the variable name
- All three declarations below are equivalent:

  ```
  int *ptr;
  int* ptr;
  int * ptr;
  ```

- Convention: Most C programmers use `int *ptr` style

Multiple Pointer Declaration:

```
int *p1, *p2, *p3;    // Three pointers to int
int *p1, p2, *p3;     // p1 and p3 are pointers, p2 is int
int* p1, p2, p3;      // Only p1 is pointer! p2 and p3 are int
```

2.2 Pointer Operators

There are two main operators for working with pointers:

Operator	Name	Description	Example

`&`	Address-of	Gets the memory address of a variable	`&variable`
`*`	Dereference	Accesses the value at the address stored in pointer	`*pointer`

2.3 Initializing Pointers

Method 1: Initialize with address of existing variable

```
int num = 42;
int *ptr = &num;  // ptr now points to num
```

Method 2: Initialize to NULL

```
int *ptr = NULL;  // Pointer points to nothing (safe initialization)
```

Method 3: Uninitialized (DANGEROUS)

```
int *ptr;  // Contains garbage value - DO NOT USE until initialized!
```

Visual Representation:

```
Memory Layout:

Variable: num = 42
Address:  0x1000
          ┌──────┐
0x1000:   │  42  │  num
          └──────┘

Variable: ptr
Address:  0x2000
          ┌────────┐
0x2000:   │ 0x1000 │  ptr (points to num)
          └────────┘
```

2.4 Using Pointers - The & and * Operators

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

```c
int main() {
    int num = 100;
    int *ptr;

    ptr = &num;  // Store address of num in ptr

    printf("Value of num: %d\n", num);           // Direct access
    printf("Address of num: %p\n", (void*)&num); // Address of num
    printf("Value of ptr: %p\n", (void*)ptr);    // Address stored in ptr
    printf("Value pointed to by ptr: %d\n", *ptr); // Dereference ptr

    // Modify through pointer
    *ptr = 200;

    printf("\nAfter *ptr = 200:\n");
    printf("Value of num: %d\n", num);           // num changed!
    printf("Value pointed to by ptr: %d\n", *ptr);

    return 0;
}

/* Output:
Value of num: 100
Address of num: 0x7ffd5c8e4a3c
Value of ptr: 0x7ffd5c8e4a3c
Value pointed to by ptr: 100

After *ptr = 200:
Value of num: 200
Value pointed to by ptr: 200
*/
```

Understanding the Operations:

```c
int num = 42;
int *ptr = &num;

// These are equivalent:
num = 100;     // Direct modification
*ptr = 100;    // Indirect modification through pointer
```

```
// Both operations change the same memory location
```

3. Pointer Arithmetics

3. Pointer Arithmetic

Pointers can be incremented, decremented, and compared. When you perform arithmetic on pointers, the operation takes into account the size of the data type being pointed to.

3.1 Basic Pointer Arithmetic Operations

Operation	Description	Example
`ptr++`	Move to next element	`ptr = ptr + 1`
`ptr--`	Move to previous element	`ptr = ptr - 1`
`ptr + n`	Move n elements forward	`ptr = ptr + 3`
`ptr - n`	Move n elements backward	`ptr = ptr - 2`
`ptr2 - ptr1`	Distance between pointers	Number of elements

3.2 How Pointer Arithmetic Works

When you add 1 to a pointer, it doesn't increase by 1 byte—it increases by the size of the data type:

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int *ptr = arr;  // Points to first element

    printf("Address of ptr: %p, Value: %d\n", (void*)ptr, *ptr);

    ptr++;  // Move to next integer (adds sizeof(int) bytes)
    printf("Address of ptr: %p, Value: %d\n", (void*)ptr, *ptr);

    ptr++;  // Move to next integer
    printf("Address of ptr: %p, Value: %d\n", (void*)ptr, *ptr);
```

```
    return 0;
}

/* Output (addresses will vary):
Address of ptr: 0x7ffd5c8e4a20, Value: 10
Address of ptr: 0x7ffd5c8e4a24, Value: 20  (increased by 4 bytes for int)
Address of ptr: 0x7ffd5c8e4a28, Value: 30  (increased by 4 bytes again)
*/
```

Memory Layout:

```
Array: arr[] = {10, 20, 30, 40, 50}

        ┌────┬────┬────┬────┬────┐
        | 10 | 20 | 30 | 40 | 50 |
        └────┴────┴────┴────┴────┘

         ↑    ↑    ↑
       ptr  ptr+1 ptr+2
      0x100 0x104 0x108  (assuming 4-byte int)
```

3.3 Pointer Arithmetic Examples

```
#include <stdio.h>

int main() {
    int numbers[] = {100, 200, 300, 400, 500};
    int *ptr = numbers;

    // Access elements using pointer arithmetic
    printf("First element: %d\n", *ptr);           // 100
    printf("Second element: %d\n", *(ptr + 1));    // 200
    printf("Third element: %d\n", *(ptr + 2));     // 300
    printf("Fifth element: %d\n", *(ptr + 4));     // 500

    // Equivalent array notation
    printf("\nUsing array notation:\n");
    printf("First element: %d\n", ptr[0]);         // 100
    printf("Second element: %d\n", ptr[1]);        // 200
    printf("Third element: %d\n", ptr[2]);         // 300
```

```
    // Distance between pointers
    int *start = &numbers[0];
    int *end = &numbers[4];
    printf("\nDistance between pointers: %ld elements\n", end - start);

    return 0;
}
```

3.4 Valid and Invalid Pointer Operations

Valid Operations:

```
int arr[5];
int *ptr = arr;

ptr++;           // Valid: increment pointer
ptr--;           // Valid: decrement pointer
ptr = ptr + 3;   // Valid: add integer to pointer
ptr = ptr - 2;   // Valid: subtract integer from pointer
int diff = ptr2 - ptr1;  // Valid: subtract two pointers (same type)
if (ptr1 < ptr2) { }     // Valid: compare pointers
```

Invalid Operations:

```
ptr = ptr * 2;    // INVALID: cannot multiply pointers
ptr = ptr / 2;    // INVALID: cannot divide pointers
ptr = ptr + ptr2; // INVALID: cannot add two pointers
```

4. Pointers and Arrays

Arrays and pointers have a very close relationship in C. In many contexts, an array name acts as a pointer to its first element.

4.1 Array Name as Pointer

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    // Array name is a pointer to first element
    printf("Address of arr: %p\n", (void*)arr);
    printf("Address of arr[0]: %p\n", (void*)&arr[0]);
    printf("These addresses are the same!\n\n");

    // Access elements using pointer notation
    printf("arr[0] = %d, *arr = %d\n", arr[0], *arr);
    printf("arr[1] = %d, *(arr+1) = %d\n", arr[1], *(arr + 1));
    printf("arr[2] = %d, *(arr+2) = %d\n", arr[2], *(arr + 2));

    return 0;
}
```

4.2 Relationship Between Arrays and Pointers

Key Equivalences:

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;

// These are equivalent:
arr[i]  ≡  *(arr + i)
arr[i]  ≡  ptr[i]
arr[i]  ≡  *(ptr + i)
&arr[i] ≡  (arr + i)
```

```
&arr[i] ≡  (ptr + i)
```

Visual Representation:

```
Array: arr[5] = {10, 20, 30, 40, 50}

Index:     0    1    2    3    4
          ┌────┬────┬────┬────┬────┐
arr -->   | 10 | 20 | 30 | 40 | 50 |
          └────┴────┴────┴────┴────┘

           ↑
         arr, &arr[0], arr+0, *(arr+0)
               ↑
           arr+1, &arr[1], *(arr+1)
                    ↑
               arr+2, &arr[2], *(arr+2)
```

4.3 Traversing Arrays with Pointers

Method 1: Using array indexing

```
int arr[5] = {10, 20, 30, 40, 50};

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

Method 2: Using pointer arithmetic

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;

for (int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
}
```

Method 3: Incrementing pointer

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;
```

```
int *end = arr + 5;

while (ptr < end) {
    printf("%d ", *ptr);
    ptr++;
}
```

4.4 Important Difference: Array vs Pointer

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;

// This is VALID:
ptr = ptr + 1;  // ptr can be modified
ptr++;          // ptr can be incremented

// This is INVALID:
arr = arr + 1;  // ERROR! Array name is a constant pointer
arr++;          // ERROR! Cannot modify array name

// However, this is valid:
int *ptr2 = arr + 1;  // Create new pointer pointing to arr[1]
```

Key Difference:

- `arr` is a **constant pointer** (cannot be reassigned)
- `ptr` is a **pointer variable** (can be modified)

5. Pointers and Functions

Pointers are essential for functions to modify variables from the calling code and to work efficiently with arrays.

5.1 Pass by Value vs Pass by Reference

Pass by Value (Without Pointers):

```
#include <stdio.h>

void tryToChange(int x) {
    x = 100;  // Only changes local copy
    printf("Inside function: x = %d\n", x);
}

int main() {
    int num = 50;
    printf("Before function: num = %d\n", num);
    tryToChange(num);
    printf("After function: num = %d\n", num);  // num unchanged!
    return 0;
}

/* Output:
Before function: num = 50
Inside function: x = 100
After function: num = 50
*/
```

Pass by Reference (With Pointers):

```
#include <stdio.h>

void actuallyChange(int *x) {
    *x = 100;  // Changes the original variable
    printf("Inside function: *x = %d\n", *x);
}
```

```
int main() {
    int num = 50;
    printf("Before function: num = %d\n", num);
    actuallyChange(&num);  // Pass address
    printf("After function: num = %d\n", num);  // num changed!
    return 0;
}

/* Output:
Before function: num = 50
Inside function: *x = 100
After function: num = 100
*/
```

5.2 Why scanf() Requires &

Now we understand why `scanf()` needs the `&` operator:

```
int age;
scanf("%d", &age);  // Pass address so scanf can modify age

// What happens inside scanf (simplified):
void scanf(const char *format, int *ptr) {
    // Read input value
    int value = /* read from keyboard */;
    *ptr = value;  // Store in the address provided
}
```

5.3 Functions Returning Multiple Values

Since C functions can only return one value, pointers allow us to "return" multiple values:

```
#include <stdio.h>

// Function to find both quotient and remainder
void divide(int dividend, int divisor, int *quotient, int *remainder) {
    *quotient = dividend / divisor;
    *remainder = dividend % divisor;
```

```
}

int main() {
    int a = 17, b = 5;
    int q, r;

    divide(a, b, &q, &r);

    printf("%d divided by %d:\n", a, b);
    printf("Quotient: %d\n", q);
    printf("Remainder: %d\n", r);

    return 0;
}
```

5.4 Swapping Values Using Pointers

A classic example of pointer usage:

```
#include <stdio.h>

// WRONG: This doesn't swap the original variables
void wrongSwap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}

// CORRECT: This swaps the original variables
void correctSwap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 10, b = 20;

    printf("Before swap: a = %d, b = %d\n", a, b);
```

```
    wrongSwap(a, b);
    printf("After wrongSwap: a = %d, b = %d\n", a, b);  // No change

    correctSwap(&a, &b);
    printf("After correctSwap: a = %d, b = %d\n", a, b);  // Swapped!

    return 0;
}
```

5.5 Passing Arrays to Functions

When passing arrays to functions, you're actually passing a pointer:

```
#include <stdio.h>

// These function declarations are equivalent:
void printArray1(int arr[], int size);
void printArray2(int *arr, int size);

void printArray1(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    printArray1(numbers, size);

    return 0;
}
```

Important Note:

```
void function(int arr[]) {
    // Inside function, sizeof(arr) gives size of pointer, not array!
    int size = sizeof(arr);  // This gives 8 (size of pointer on 64-bit)
```

```
    // WRONG! Does not give array size
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);  // This is correct
    function(numbers);
    return 0;
}
```

6. Pointers and Strings

In C, strings are arrays of characters, so pointers work naturally with strings.

6.1 String Representation

```
char str1[] = "Hello";     // Array notation
char *str2 = "Hello";      // Pointer notation

// Both represent the same thing in memory:
// 'H' 'e' 'l' 'l' 'o' '\0'
```

Memory Layout:

```
str1: char array (modifiable)
┌───┬───┬───┬───┬───┬────┐
| H | e | l | l | o | \0 |
└───┴───┴───┴───┴───┴────┘


str2: pointer to string literal (read-only)
     ┌────────┐
str2 | addr   |─────> "Hello\0" (in read-only memory)
     └────────┘
```

6.2 String Traversal Using Pointers

```
#include <stdio.h>

void printString(char *str) {
    while (*str != '\0') {  // Until null terminator
        printf("%c", *str);
        str++;  // Move to next character
    }
    printf("\n");
}

int main() {
```

```
    char message[] = "Hello, World!";
    printString(message);
    return 0;
}
```

6.3 String Length Using Pointers

```
#include <stdio.h>

int stringLength(char *str) {
    int length = 0;
    while (*str != '\0') {
        length++;
        str++;
    }
    return length;
}

// Alternative using pointer arithmetic
int stringLength2(char *str) {
    char *start = str;
    while (*str != '\0') {
        str++;
    }
    return str - start;  // Pointer subtraction
}

int main() {
    char text[] = "Programming";
    printf("Length: %d\n", stringLength(text));
    return 0;
}
```

6.4 String Copy Using Pointers

```
#include <stdio.h>

void stringCopy(char *dest, char *src) {
    while (*src != '\0') {
```

```
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';  // Don't forget null terminator!
}

// More elegant version
void stringCopy2(char *dest, char *src) {
    while ((*dest++ = *src++));  // Copy until '\0' (which is 0/false)
}

int main() {
    char source[] = "Hello";
    char destination[50];

    stringCopy(destination, source);
    printf("Copied string: %s\n", destination);

    return 0;
}
```

7. Dynamic Memory Allocation & Array

7.1 Introduction to Dynamic Memory

Up until now, we've used **static memory allocation** where array sizes must be known at compile time:

```
int arr[100];  // Size fixed at compile time
```

Problems with static allocation:

- Waste memory if you allocate too much
- Run out of space if you allocate too little
- Cannot adjust size during program execution

Dynamic memory allocation solves these problems by allowing you to allocate memory at runtime using special functions.

7.2 Memory Layout in C

```
┌─────────────────────────────────┐
│         Stack                   │  ← Local variables, function calls
│  (grows downward)               │     Automatically managed
│            ↓                    │
├─────────────────────────────────┤
│                                 │
│         (free space)            │
│                                 │
│            ↑                    │
├─────────────────────────────────┤
│         Heap                    │  ← Dynamic memory allocation
│  (grows upward)                 │     Manually managed (malloc/free)
├─────────────────────────────────┤
│    Data Segment                 │  ← Global and static variables
├─────────────────────────────────┤
│    Code Segment                 │  ← Program instructions
```

```
└──────────────────────────────┘
```

7.3 Dynamic Memory Functions

C provides four main functions for dynamic memory management (defined in `<stdlib.h>`):

Function	Purpose	Syntax
`malloc()`	Allocates memory	`void* malloc(size_t size)`
`calloc()`	Allocates and initializes to zero	`void* calloc(size_t n, size_t size)`
`realloc()`	Resizes allocated memory	`void* realloc(void* ptr, size_t size)`
`free()`	Releases allocated memory	`void free(void* ptr)`

7.4 malloc() - Memory Allocation

Purpose: Allocates a block of memory of specified size (in bytes)

Syntax:

```
void* malloc(size_t size);
```

Returns:

- Pointer to allocated memory on success
- `NULL` if allocation fails

Example:

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

int main() {
    int *ptr;
    int n = 5;

    // Allocate memory for 5 integers
    ptr = (int*)malloc(n * sizeof(int));

    // Check if allocation was successful
    if (ptr == NULL) {
```

```
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Use the allocated memory
    for (int i = 0; i < n; i++) {
        ptr[i] = i * 10;
    }

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

    // Free the allocated memory
    free(ptr);
    ptr = NULL;  // Good practice

    return 0;
}
```

Important Notes:

- Always multiply by `sizeof(data_type)` to get correct byte size
- Always check if `malloc()` returns `NULL`
- Always cast the return value: `(int*)malloc(...)`
- Memory allocated by `malloc()` contains **garbage values**

7.5 calloc() - Contiguous Allocation

Purpose: Allocates memory and initializes all bytes to zero

Syntax:

```
void* calloc(size_t n, size_t size);
```

Parameters:

- `n`: Number of elements
- `size`: Size of each element

Example:

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

int main() {
    int *ptr;
    int n = 5;

    // Allocate and initialize memory for 5 integers
    ptr = (int*)calloc(n, sizeof(int));

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

    // Print values (all will be 0)
    printf("Initial values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    free(ptr);
    ptr = NULL;

    return 0;
}

/* Output:
Initial values: 0 0 0 0 0
*/
```

malloc() vs calloc():

Feature	malloc()	calloc()
Parameters	1 (total bytes)	2 (number, size)
Initialization	Garbage values	All zeros

Feature	malloc()	calloc()
Speed	Faster	Slightly slower
Use case	When you'll initialize values	When you need zero-initialized memory

7.6 realloc() - Resize Memory

Purpose: Changes the size of previously allocated memory

Syntax:

```
void* realloc(void* ptr, size_t new_size);
```

Example:

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

int main() {
    int *ptr;
    int n = 5;

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

    // Fill initial array
    for (int i = 0; i < n; i++) {
        ptr[i] = i + 1;
    }

    printf("Initial array (size %d): ", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
```

```
    // Resize to 10 elements
    n = 10;
    ptr = (int*)realloc(ptr, n * sizeof(int));

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

    // Fill new elements
    for (int i = 5; i < n; i++) {
        ptr[i] = i + 1;
    }

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

    free(ptr);
    ptr = NULL;

    return 0;
}

/* Output:
Initial array (size 5): 1 2 3 4 5
Resized array (size 10): 1 2 3 4 5 6 7 8 9 10
*/
```

Important Notes about realloc():

- If `new_size` is larger, existing data is preserved, new space is uninitialized
- If `new_size` is smaller, data is truncated
- May move the block to a new location (address may change)
- If realloc fails, original pointer remains valid
- If `ptr` is `NULL`, behaves like `malloc()`

7.7 free() - Deallocate Memory

Purpose: Releases memory back to the system

Syntax:

```
void free(void* ptr);
```

Example:

```
int *ptr = (int*)malloc(100 * sizeof(int));

// Use the memory...

free(ptr);     // Release memory
ptr = NULL;    // Set to NULL to avoid dangling pointer
```

Important Rules:

1. Only free memory that was allocated with malloc/calloc/realloc
2. Free each block exactly once
3. Don't use memory after freeing it
4. Set pointer to NULL after freeing (good practice)

7.8 Dynamic Arrays - Complete Example

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

int main() {
    int n;
    int *arr;

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

    // Allocate memory dynamically
    arr = (int*)malloc(n * sizeof(int));

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

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

    // Process: find sum and average
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    double average = (double)sum / n;

    // Output results
    printf("\nArray elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nSum: %d\n", sum);
    printf("Average: %.2f\n", average);

    // Free allocated memory
    free(arr);
    arr = NULL;

    return 0;
}
```

7.9 Dynamic 2D Arrays

Method 1: Array of Pointers (Rows can have different lengths)

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

int main() {
    int rows = 3, cols = 4;
    int **matrix;
```

```
    // Allocate array of row pointers
    matrix = (int**)malloc(rows * sizeof(int*));

    // Allocate each row
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }

    // Fill matrix
    int value = 1;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = value++;
        }
    }

    // Print matrix
    printf("Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%3d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Free memory
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
    matrix = NULL;

    return 0;
}
```

Method 2: Single Contiguous Block

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

```c
int main() {
    int rows = 3, cols = 4;
    int *matrix;

    // Allocate as single block
    matrix = (int*)malloc(rows * cols * sizeof(int));

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

    // Fill matrix using formula: matrix[i][j] = matrix[i * cols + j]
    int value = 1;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i * cols + j] = value++;
        }
    }

    // Print matrix
    printf("Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%3d ", matrix[i * cols + j]);
        }
        printf("\n");
    }

    // Free memory (single free needed)
    free(matrix);
    matrix = NULL;

    return 0;
}
```

7.10 Dynamic Memory Best Practices

1. Always Check for NULL

```
int *ptr = (int*)malloc(n * sizeof(int));
if (ptr == NULL) {
    fprintf(stderr, "Memory allocation failed!\n");
    exit(1);
}
```

2. Always Free Allocated Memory

```
// Allocate
int *data = (int*)malloc(100 * sizeof(int));

// Use data...

// Free when done
free(data);
data = NULL;
```

3. Don't Double Free

```
int *ptr = (int*)malloc(10 * sizeof(int));
free(ptr);
free(ptr);  // ERROR! Double free - undefined behavior
```

4. Don't Use After Free

```
int *ptr = (int*)malloc(10 * sizeof(int));
free(ptr);
ptr[0] = 5;  // ERROR! Using freed memory
```

5. Match Every malloc with free

```
void function() {
    int *ptr = (int*)malloc(100 * sizeof(int));
    // Use ptr...
    free(ptr);  // Don't forget!
}
```

7.11 Memory Leaks

A **memory leak** occurs when allocated memory is not freed:

Example of Memory Leak:

```
void badFunction() {
    int *ptr = (int*)malloc(1000 * sizeof(int));
    // Use ptr...
    return;  // BUG! Memory not freed - leaked!
}

int main() {
    for (int i = 0; i < 1000; i++) {
        badFunction();  // Leaks memory every iteration
    }
    return 0;
}
```

Fixed Version:

```
void goodFunction() {
    int *ptr = (int*)malloc(1000 * sizeof(int));
    // Use ptr...
    free(ptr);  // Properly freed
    return;
}
```

Another Common Leak:

```
int *ptr = (int*)malloc(100 * sizeof(int));
ptr = (int*)malloc(200 * sizeof(int));  // LEAK! Lost reference to first block
```

Fixed:

```
int *ptr = (int*)malloc(100 * sizeof(int));
free(ptr);  // Free first
ptr = (int*)malloc(200 * sizeof(int));  // Then allocate new
```

8. Common Pointer Pitfalls and Best Practices

8.1 Uninitialized Pointers

WRONG:

```
int *ptr;      // Uninitialized - contains garbage address
*ptr = 42;     // DANGER! Writing to unknown memory location
               // May cause segmentation fault
```

CORRECT:

```
int *ptr = NULL;   // Initialize to NULL
int num = 0;
ptr = &num;        // Assign valid address before use
*ptr = 42;         // Now safe to dereference
```

8.2 Dangling Pointers

A **dangling pointer** points to memory that has been freed or is no longer valid:

WRONG:

```
int *ptr;
{
    int num = 42;
    ptr = &num;
}  // num goes out of scope here
// ptr is now dangling - points to invalid memory
printf("%d", *ptr);  // DANGER! Undefined behavior
```

CORRECT:

```
int num = 42;
int *ptr = &num;
// Use ptr while num is in scope
```

```
printf("%d", *ptr);  // Safe
```

Dangling Pointer with free():

```
int *ptr = (int*)malloc(sizeof(int));
*ptr = 42;
free(ptr);
// ptr is now dangling
printf("%d", *ptr);  // DANGER! Undefined behavior

// Better:
free(ptr);
ptr = NULL;  // Set to NULL after freeing
if (ptr != NULL) {
    printf("%d", *ptr);  // This check prevents the error
}
```

8.3 NULL Pointer Dereference

WRONG:

```
int *ptr = NULL;
*ptr = 42;  // CRASH! Cannot dereference NULL pointer
```

CORRECT:

```
int *ptr = NULL;

if (ptr != NULL) {  // Always check before dereferencing
    *ptr = 42;
} else {
    printf("Error: NULL pointer\n");
}
```

8.4 Array Bounds with Pointers

WRONG:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
```

```
int value = *(ptr + 10);  // Out of bounds! Undefined behavior
```

CORRECT:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
int size = 5;

for (int i = 0; i < size; i++) {
    printf("%d ", *(ptr + i));  // Safe: within bounds
}
```

8.5 Returning Pointer to Local Variable

WRONG:

```
int* createNumber() {
    int num = 42;
    return &num;  // DANGER! num is destroyed after function returns
}

int main() {
    int *ptr = createNumber();
    printf("%d", *ptr);  // Undefined behavior - dangling pointer
    return 0;
}
```

CORRECT - Using Dynamic Allocation:

```
int* createNumber() {
    int *num = (int*)malloc(sizeof(int));
    *num = 42;
    return num;  // Safe - memory persists
}

int main() {
    int *ptr = createNumber();
    printf("%d", *ptr);
    free(ptr);  // Don't forget to free!
    return 0;
```

```
}
```

CORRECT - Using Static Variable:

```
int* createNumber() {
    static int num = 42;  // Static - persists after function returns
    return &num;
}
```

8.6 Best Practices Summary

1. **Always initialize pointers**

   ```
   int *ptr = NULL;  // Good
   int *ptr;         // Bad
   ```

2. **Check for NULL before dereferencing**

   ```
   if (ptr != NULL) {
       *ptr = value;
   }
   ```

3. **Set pointers to NULL after freeing**

   ```
   free(ptr);
   ptr = NULL;
   ```

4. **Be careful with pointer arithmetic**

   ```
   // Ensure you don't go out of array bounds
   if (ptr + i < arr + size) {
       // Safe to access
   }
   ```

5. **Use const for pointers that shouldn't modify data**

   ```
   void printString(const char *str) {
       // str cannot be used to modify the string
   }
   ```

6. **Match every malloc with free**

   ```
   int *ptr = (int*)malloc(100 * sizeof(int));
   // Use ptr...
   ```

```
free(ptr);
```

9. Practical Examples with Dynamic Memory

9.1 Dynamic Array with User Input

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

int main() {
    int n, *arr;

    printf("How many numbers do you want to enter? ");
    scanf("%d", &n);

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

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

    // Find min and max
    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }

    printf("Minimum: %d\n", min);
```

```
    printf("Maximum: %d\n", max);

    free(arr);
    return 0;
}
```

9.2 Growing Array (Dynamic Resize)

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

int main() {
    int capacity = 2;
    int size = 0;
    int *arr;
    int input;

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

    printf("Enter integers (enter -1 to stop):\n");

    while (1) {
        scanf("%d", &input);
        if (input == -1) break;

        // Check if we need more space
        if (size >= capacity) {
            capacity *= 2;  // Double the capacity
            arr = (int*)realloc(arr, capacity * sizeof(int));
            if (arr == NULL) {
                printf("Memory reallocation failed!\n");
                return 1;
            }
            printf("Array resized to capacity: %d\n", capacity);
```

```
        }

        arr[size++] = input;
    }

    // Print results
    printf("\nYou entered %d numbers:\n", size);
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

9.3 Dynamic String Operations

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

char* concatenateStrings(const char *s1, const char *s2) {
    int len1 = strlen(s1);
    int len2 = strlen(s2);

    // Allocate memory for result
    char *result = (char*)malloc((len1 + len2 + 1) * sizeof(char));

    if (result == NULL) {
        return NULL;
    }

    // Copy first string
    strcpy(result, s1);
    // Concatenate second string
    strcat(result, s2);

    return result;
```

```
}

int main() {
    char str1[] = "Hello, ";
    char str2[] = "World!";

    char *combined = concatenateStrings(str1, str2);

    if (combined != NULL) {
        printf("Result: %s\n", combined);
        free(combined);
    }

    return 0;
}
```

9.4 Remove Duplicates from Dynamic Array

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

int* removeDuplicates(int *arr, int size, int *newSize) {
    // Allocate worst-case size (all unique)
    int *result = (int*)malloc(size * sizeof(int));
    if (result == NULL) {
        return NULL;
    }

    int count = 0;

    for (int i = 0; i < size; i++) {
        int isDuplicate = 0;

        // Check if current element already exists in result
        for (int j = 0; j < count; j++) {
            if (arr[i] == result[j]) {
                isDuplicate = 1;
                break;
            }
```

```
        }

        if (!isDuplicate) {
            result[count++] = arr[i];
        }
    }

    // Resize to actual size needed
    result = (int*)realloc(result, count * sizeof(int));
    *newSize = count;

    return result;
}

int main() {
    int arr[] = {1, 2, 3, 2, 4, 1, 5, 3, 6};
    int size = sizeof(arr) / sizeof(arr[0]);
    int newSize;

    printf("Original array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    int *unique = removeDuplicates(arr, size, &newSize);

    if (unique != NULL) {
        printf("Array without duplicates: ");
        for (int i = 0; i < newSize; i++) {
            printf("%d ", unique[i]);
        }
        printf("\n");

        free(unique);
    }

    return 0;
}
```

9.5 Dynamic Matrix Operations

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

// Allocate matrix
int** createMatrix(int rows, int cols) {
    int **matrix = (int**)malloc(rows * sizeof(int*));
    if (matrix == NULL) return NULL;

    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
        if (matrix[i] == NULL) {
            // Free already allocated rows
            for (int j = 0; j < i; j++) {
                free(matrix[j]);
            }
            free(matrix);
            return NULL;
        }
    }

    return matrix;
}

// Free matrix
void freeMatrix(int **matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Multiply two matrices
int** multiplyMatrices(int **A, int **B, int r1, int c1, int c2) {
    int **result = createMatrix(r1, c2);
    if (result == NULL) return NULL;

    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
```

```
            result[i][j] = 0;
            for (int k = 0; k < c1; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return result;
}

int main() {
    int r1 = 2, c1 = 3, c2 = 2;

    // Create matrices
    int **A = createMatrix(r1, c1);
    int **B = createMatrix(c1, c2);

    // Fill matrix A
    printf("Matrix A:\n");
    int val = 1;
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c1; j++) {
            A[i][j] = val++;
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }

    // Fill matrix B
    printf("\nMatrix B:\n");
    val = 1;
    for (int i = 0; i < c1; i++) {
        for (int j = 0; j < c2; j++) {
            B[i][j] = val++;
            printf("%d ", B[i][j]);
        }
        printf("\n");
    }

    // Multiply
    int **C = multiplyMatrices(A, B, r1, c1, c2);
```

```
    printf("\nMatrix C (A × B):\n");
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    // Free all matrices
    freeMatrix(A, r1);
    freeMatrix(B, c1);
    freeMatrix(C, r1);

    return 0;
}
```